

AD-A165 345

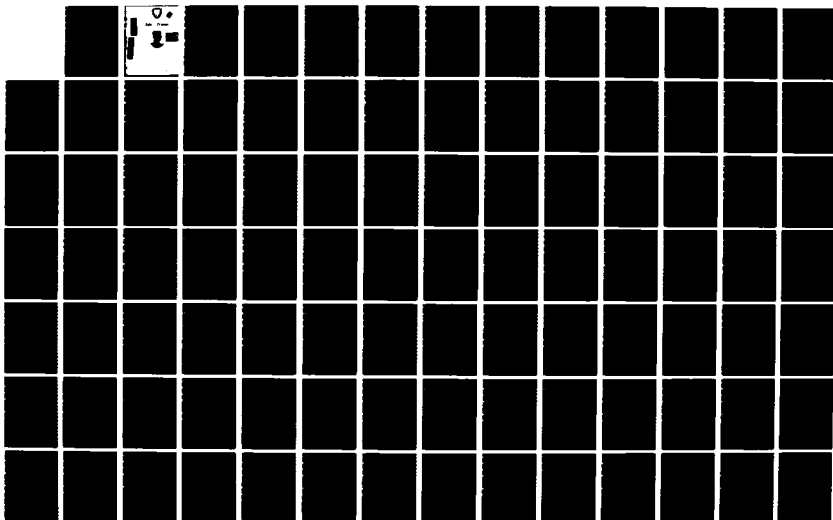
ADA (TRADEMARK) PRIMER(U) SOFTECH INC WALTHAM MA 1986  
DAB07-83-C-K506

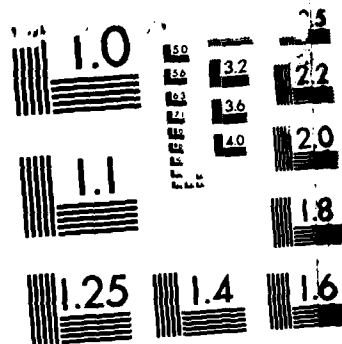
1/5

UNCLASSIFIED

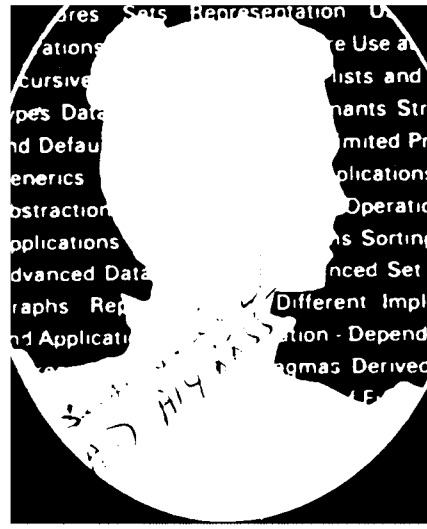
F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A



## TABLE OF CONTENTS

<u>Section</u>		<u>Page</u>
1	OVERVIEW OF THE ADA LANGUAGE	1-1
2	INTRODUCTION TO PROGRAM UNITS	2-1
	2.1 Packages	2-3
	2.2 Procedures	2-15
	2.3 Functions	2-29
	2.4 Parameter Modes	2-41
	2.5 Parameter Notation	2-53
	2.6 Program Structure	2-61
3	LEXICAL ELEMENTS	3-1
	3.1 Lexical Style	3-3
	3.2 Identifiers	3-13
4	INTRODUCTION TO DATA	4-1
	4.1 Predefined Integer	4-3
	4.2 Integer Literals	4-13
	4.3 Predefined Float	4-19
	4.4 Real Literals	4-27
	4.5 Type Conversion	4-33
5	ENUMERATION: TYPES AND CONTROL STRUCTURES	5-1
	5.1 Enumeration Types/Objects	5-3
	5.2 Case Statement	5-13
	5.3 Boolean Types/Objects	5-25
	5.4 If Statement	5-33



## TABLE OF CONTENTS (continued)

<u>Section</u>		<u>Page</u>
6	NUMERIC TYPES	6-1
	6.1 User Defined Integer Types/Objects	6-3
	6.2 User Defined Real Types/Objects	6-11
	6.3 Subtypes	6-19
7	ADVANCED FEATURES OF SCALAR TYPES	7-1
	7.1 Short Circuit Control Form	7-3
	7.2 Case Statement with Numerics	7-9
	7.3 Attributes	7-15
8	ARRAY TYPES AND ITERATIVE CONTROL STRUCTURES	8-1
	8.1 Constrained Arrays	8-3
	8.2 Loops	8-13
	8.3 Unconstrained Arrays (String)	8-25
	8.4 Array of Arrays/Array of String	8-37
9	RECORD TYPES	9-1
	9.1 Record Types/Objects	9-3
	9.2 Records of Arrays	9-11
	9.3 Arrays of Records	9-19
	9.4 Discriminants	9-27
10	ACCESS TYPES	10-1
	10.1 Access Types and Allocators	10-3
11	PROGRAM STRUCTURE AND SEPARATE COMPILATION	11-1
	11.1 Subunits/Library Units	11-3

(1)

## PREFACE

This workbook was developed under Contract Number DAAB07-83-C-K514 for the U.S. Army, Center for Tactical Computer Systems (CENTACS), Fort Monmouth, New Jersey.

Ada is a high order language (HOL) developed by the Department of Defense (DoD) to be used in programming embedded systems. It is anticipated that use of Ada will significantly reduce the costs associated with defense software by providing more maintainable code. One of the goals of this workbook is to illustrate how to use the features of the Ada language to help produce more maintainable code.

Ada Primer is the first of a series of three workbooks. The other two workbooks are entitled respectively, Advanced Ada and Real-Time Ada. Ada Primer leads the novice through the fundamentals of Ada by addressing the so-called Pascal subset of Ada. This workbook may be used in conjunction with module L202, Basic Ada Programming. This module is one element of the U. S. Army Ada Training Curriculum which was developed under Contract No. DAAK80-80-C-0187. A student, having completed L202 in conjunction with this workbook, shall be able to use an Ada package and code Ada subprograms using the Pascal subset of Ada. Advanced Ada, which may be used in conjunction with module L305, Algorithms and Data Structures in Ada, addresses additional features of Ada. Advanced Ada emphasizes program structure in Ada. At the completion of Advanced Ada the student shall be able to code complex Ada programs. Real-Time Ada may be used in conjunction with module L401, Real Time Systems in Ada. Real-Time Ada emphasizes the real time aspects of Ada and places heavy emphasis on Ada's tasking mechanism. At the completion of Real-Time Ada the student will be prepared to handle more complex system implementation and design issues.

With the exception of Section 1 of this workbook, each workbook is organized as a series of Ada exercises to be solved by the reader. Taken sequentially as presented in the workbook, the exercises expose the student to the features of Ada in a logical fashion. In order to gain full benefit from this workbook, it is strongly recommended that the exercises be completed in the order presented. The solution to a specific exercise is located directly following the exercise.

Each exercise consists of the following:

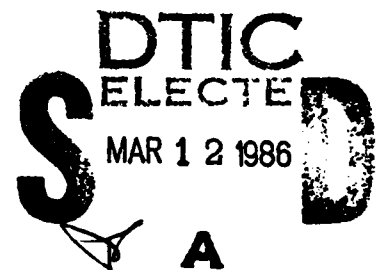
- Exercise Portion
  - Objective
  - Tutorial
  - Problem

*Supervised  
AD-A148855*

This document has been approved  
for public release and sale; its  
distribution is unlimited.

1110-1-1.1

- 1



86 3 11 145

- Solution Portion

- Solution
- Rationale for the Solution
- Alternative Solution
- Discussion

In the exercise portion, the objective states what is to be illustrated or taught in the exercise. The tutorial is basically textual material designed to teach the student the information stated in the objective. The statement of the problem is last and is designed to test the student's knowledge of the topic presented. The problem format varies from filling in a blank to coding an entire subprogram.

Four components constitute the solution portion. First a solution is presented. The rationale for the solution gives the reasoning behind the choice of the solution presented. An alternative solution presents another way of approaching the problem and finally, the two solutions are compared. It must be realized that there is not always an alternative solution. If that is the case for a specific exercise it is explicitly stated in the text.

As previously stated Section 1 of this workbook is different. It discusses a very simple Ada procedure, giving the reader an initial look at an Ada program and a frame of reference for the remainder of the workbook.

This workbook is intended for those individuals who have either a Bachelor's Degree in one of the sciences (with Computer Science courses included), an Associate in Arts Degree in Computer Science, or a strong background in high order languages such as Pascal, FORTRAN, JOVIAL, or CMS-2.

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A1	

## TABLE OF CONTENTS (continued)

<u>Section</u>		<u>Page</u>
12	USING LIBRARY UNITS	12-1
	12.1 Scope and Visibility	12-3
	12.2 Overloading	12-15
	12.3 Private Types	12-21
	12.4 Generics	12-29
13	PACKAGES	13-1
	13.1 Interface vs Implementation	13-3
14	EXCEPTIONS	14-1
	14.1 Exceptions and Exception Handling	14-3
15	INPUT/OUTPUT	15-1
	15.1 Package Text_IO	15-3
16	GLOSSARY	16-1
17	CROSS REFERENCE	17-1
18	INDEX	18-1

This page intentionally left blank

# **Section 1**

## **OVERVIEW OF THE ADA LANGUAGE**

This page intentionally left blank

## OVERVIEW OF THE ADA LANGUAGE

This section introduces a very small Ada program and talks about it. This provides a frame of reference for the remainder of the workbook and answers the question, "What does an Ada program look like?"

Here is an elementary Ada program that reads a value and prints out the square root of that value.\* It reads and prints in a loop as long as the end of the file has not been reached. When the end of the file is reached, the program terminates.

```
with Math_Lib; use Math_Lib;
with Simple_IO; use Simple_IO;
procedure Square_Root is

    X : Float;

begin -- Square_Root

    while not End_Of_File
        loop
            Get (X);
            Put (Sqrt (X));
        end loop;

end Square_Root;
```

What do we see here? First notice that Ada is a free format language. Statements may start anywhere on the line. Extra spaces and blank lines are permitted. Extra spaces are used to align items vertically to improve readability (such as in the first two lines) and to indent lines to highlight program structure (as in loop ... end loop;). Blank lines can be used to highlight a certain section of code. Semicolons terminate some lines yet not others. A semicolon terminates a declaration or a statement. There is no semicolon after "loop" because that is not the end of the loop statement. The end of the loop statement is indicated by "end loop", which is why there is a semicolon there. The proper use of semicolons will become clear as Ada's features are presented.

---

\* This may cause problems with an interactive input file.



The code states that we have a procedure called `Square_Root`. An Ada procedure is analogous to a FORTRAN or Assembler subroutine. It is introduced by the word "procedure." This procedure requires access to global data provided by packages named `Math_Lib` and `Simple_IO`. Access to that data is provided by a context clause in Ada which is written

```
with Unit_Name; use Unit_Name;
```

where `Unit_Name` is the name of the unit containing the global data. The two context clauses define the context in which procedure `Square_Root` executes.

A procedure is structurally divided by the word "begin." Local data is placed before the begin and executable statements are placed after the begin. The word "end," optionally followed by the name of the procedure, terminates the procedure. The structure of `Square_Root` is depicted below.

```
procedure Square_Root is
    -- local data

begin -- Square_Root

    -- executable statements

end Square_Root;
```

As previously stated, data local to the procedure is placed prior to the word "begin." `Square_Root` has one local data item, called `X`, which is a floating point variable. This means it must have decimal values.

Unlike other languages, input and output are treated just like other facilities in Ada. Thus to do I/O in Ada, the user specifically indicates he wants to access the unit which contains the I/O operations.

The two statements

```
Get (X);  
Put (Sqrt (X));
```

are procedure calls to the procedures Get and Put located in Simple\_IO. These procedures are imported via the context clause

```
with Simple_IO; use Simple_IO;
```

Procedures are called in Ada by naming the procedure. Unlike FORTRAN or assembler, the name does not need to be prefaced by the word "CALL". The function Sqrt is imported via the context clause

```
with Math_Lib; use Math_Lib;
```

As stated in the introduction, the remainder of the workbook takes the reader through the features of the so-called Pascal subset of Ada and explores the features introduced here in much greater detail.

This page intentionally left blank

## **Section 2**

### **INTRODUCTION TO PROGRAM UNITS**

- 2.1 Packages**
- 2.2 Procedures**
- 2.3 Functions**
- 2.4 Parameter Modes**
- 2.5 Parameter Notation**
- 2.6 Program Structure**

This page intentionally left blank

## EXERCISE 2.1

### OBJECTIVE

To explain the concept of an Ada package.

### TUTORIAL

Webster's Dictionary defines a package as "something that is ... prepared in compact form" and alternatively as "any finished product which has been made ready for immediate operation ... or use by preassembling all essential elements into a self contained, compact unit which has all essential elements ready for use." Actually this is a fairly good intuitive definition of an Ada program unit called a package.

Instead of dealing with system libraries, FORTRAN COMMONS, or JOVIAL COMPOOLS, Ada supplies a program unit called a package. An Ada package is a mechanism which groups logically related resources into one "box". Just as attic storage boxes are labeled with names appropriate to their contents, e.g. Photographs, Ada packages must be named, e.g. Simple Graphics. (Notice the use of the underline in the name to facilitate readability. The rules for writing names are addressed in Exercise 3.2.)

Suppose we have a simple graphics package which provides the following resources:

- Gives us a means to write points in Cartesian Coordinates.
- Draws a line given the starting and terminating points of the line in Cartesian Coordinates.
- Draws a circle given the center in Cartesian Coordinates and the radius.

In Ada, we state the above in a package specification as follows:

```
package Simple_Graphics is

  type Point is array (1 .. 2) of Float;
  procedure Draw_Line (From : in Point; To : in Point);
  procedure Draw_Circle (Center : in Point; Radius : in Float);

end Simple_Graphics;
```

The following paragraphs discuss each of the elements of the specification for package Simple\_Graphics.

package Simple\_Graphics is

This line states that we have a specification for a package named Simple Graphics. The word "is" completes the line and indicates that the contents of the package follow. Any items appearing between this line and the line "end Simple\_Graphics;" declare the resources of the package that are available to a user.

type Point is array (1 .. 2) of Float;

A data type called Point is introduced. It is declared to be a one dimensional array with integer indices and two components of type Float. The range of the indices is from 1 to 2. Float is a predefined data type in Ada. A value of type Float must be written with a decimal point. Point will allow us to create things which look like points, e.g. (1.2, 3.0).

procedure Draw\_Line (From: in Point; To: in Point);

This declares that within package Simple\_Graphics there is a procedure named Draw\_Line. (A procedure is one of the workhorse Ada program units and is addressed in Exercise 2.2). This procedure needs the two data items listed within the parentheses, called parameters, in order to execute. The parameters represent the two points to be connected by a line. Notice the parameters are separated by semicolons. The colon, ":" separates the name of the parameter from its data type. For each parameter a type must be specified. Here the type is Point which is defined above. The final semicolon terminates the declaration. When a user wants to draw a line he "calls" Draw\_Line with the two values for the two points to be connected enclosed in parentheses. Here the parameters are called formal parameters, as we are defining what parameters the procedure requires. When a user calls the procedure with specific values, these values are called the actual parameters.

To summarize, we are declaring that package Simple\_Graphics has a procedure called Draw\_Line. This declaration states the name of the procedure, lists the necessary parameters (data items needed by the procedure) and states their respective data type.

```
procedure Draw_Circle (Center: in Point; Radius : in Float);
```

This is another resource or procedure available to the user of package Simple\_Graphics. It has two parameters, each of a different type. By calling this procedure and supplying a center point and a radius, the user is able to draw a circle.

```
end Simple_Graphics;
```

This line terminates the package specification for Simple\_Graphics. The semicolon is mandatory. The use of the package name is optional, but strongly recommended. This workbook will always include the package name after the word end.

Once access to this package is obtained, these resources are available for use. The code enclosed within the box on page 2-3 is called a package declaration. A package declaration, usually known as a specification, states what is available in the package. It does not state how those resources have been implemented. The package body shows how these resources are implemented. (See Figure 2-1(a)). Package bodies are discussed in Exercise 2.6. This package specification states that procedure Draw\_Line is an available resource. It needs two parameters of type Point to execute. We do not know how this procedure is coded or how it operates. We do not know whether it draws from right to left or from left to right. If we could see the code for Draw\_Line in the package body (see Figure 2-1(b)) we would know how the line is drawn. But we really do not need to know this information. We just need to know how to gain access to this resource and how to use it once access is gained. This package meets Webster's definition. This package specification is a compact, self contained unit enclosing ready-to-use units. It is compact because the entities enclosed are closely united in terms of subject matter; it is self contained because the code is sufficient in itself to provide a very simple graphics package; the units contained are ready for use (as long as the package body has been coded and compiled).

Remember, a package specification is like a contract. It states what is to be provided by the package. The package body implements the contract.

As an example of how to use the resources provided in package Simple Graphics, consider the following procedure, called Main Proc, which draws a line between the points (2.5, 3.5) and (4.0, 6.3).



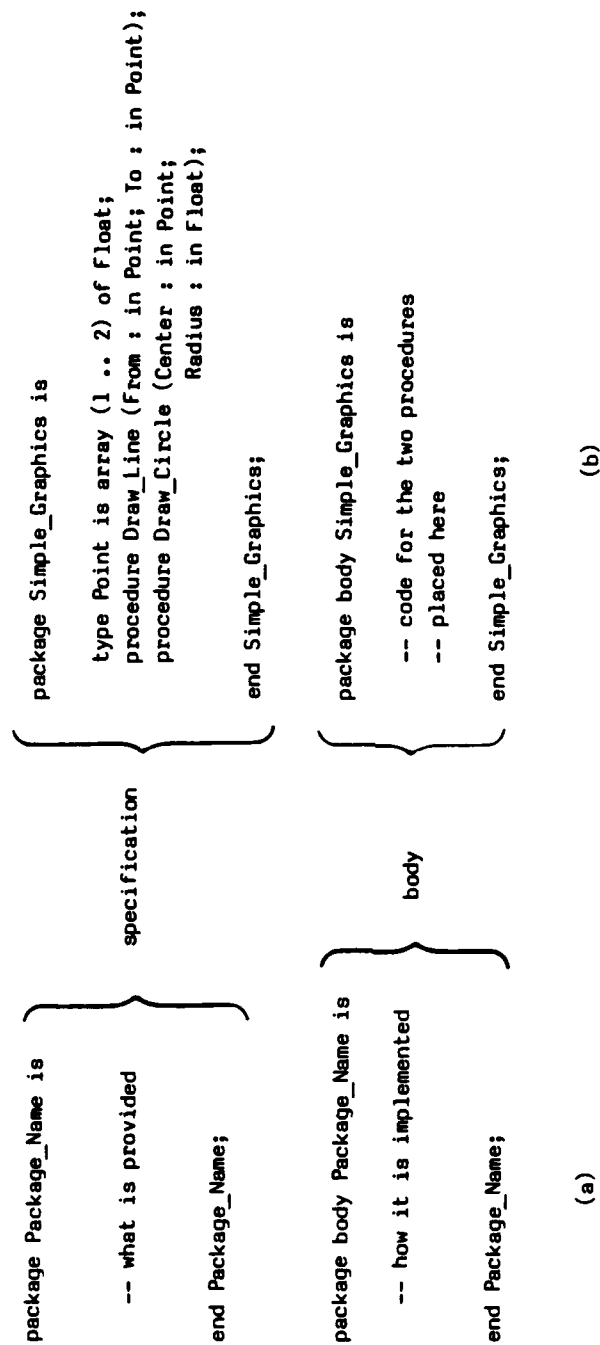


Figure 2-1. Package Structure

```

with Simple_Graphics;
procedure Main_Proc is

    -- Two hyphens indicate a comment line in Ada.
    -- A comment line gives the programmer the opportunity
    -- to document his code and place within the code
    -- information that is relevant to understanding
    -- the code but not required for the execution of
    -- the code.

begin -- Main_Proc

    -- Draw a line from (2.5, 3.5) to (4.0, 6.3)

    Simple_Graphics.Draw_Line ((2.5, 3.5), (4.0, 6.3));

end Main_Proc;

```

The context clause "with Simple\_Graphics" provides access to the package and is placed prior to the main procedure. The procedure Main\_Proc now can avail itself of the resources provided by Simple\_Graphics. Notice the call to Draw\_Line is written as Simple\_Graphics.Draw\_Line. It says to execute procedure Draw\_Line which is located in package Simple\_Graphics with the values specified. The actual parameters passed to Draw\_Line replace the formal parameters specified in the definition of Draw\_Line in the order of appearance. That means that (2.5, 3.5) is the value used for From and (4.0, 6.3) is the value used for To. In this case (2.5, 3.5) denotes a two element array with two components with values 2.5 and 3.5.

Notice that the call to Draw\_Line takes the form of the package name, directly followed by a dot, ".", directly followed by the procedure name Draw\_Line. This can be abbreviated by using a "use" clause as follows:

```
with Simple_Graphics; use Simple_Graphics;  
procedure Main_Proc is
```

```
-- Two hyphens indicate a comment line in Ada.  
-- A comment line gives the programmer the opportunity  
-- to document his code and place within the code  
-- information which is relevant to understanding  
-- the code but not required for the execution of  
-- the code.
```

```
begin -- Main_Proc
```

```
-- Draw a line from (2.5, 3.5) to (4.0, 6.3)
```

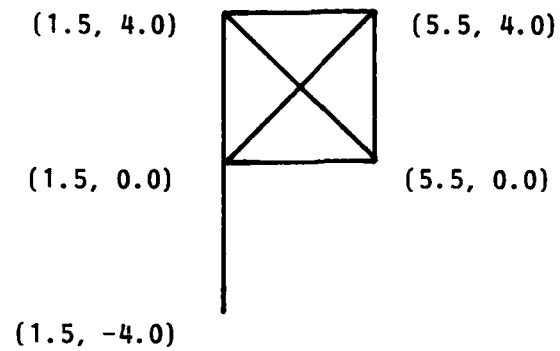
```
Draw_Line ((2.5, 3.5), (4.0, 6.3));
```

```
end Main_Proc;
```

Notice that with the addition of the "use" clause the prefix "Simple\_Graphics." is legal but no longer required in the call to Draw\_Line. The "use" clause is placed after the "with" clause.

### PROBLEM

Write a procedure called `Draw_Flag` which draws the following flag. The code should not label the diagram with the coordinates of each point.



This page intentionally left blank

## SOLUTION TO EXERCISE 2.1

### SOLUTION

```
with Simple_Graphics; use Simple_Graphics;
procedure Draw_Flag is

    -- This procedure will draw the flag
    -- by first drawing the entire flag
    -- pole from (1.5, 4.0) to (1.5, -4.0).
    -- The top, right side, and bottom
    -- of the flag will be drawn next,
    -- in the order specified. Finally
    -- the two diagonal lines are drawn.

    -- This procedure does not label the
    -- diagram with the coordinates of the points.

begin -- Draw_Flag

    -- Draw entire flag pole
    Draw_Line ((1.5, -4.0), (1.5, 4.0));

    -- Draw top
    Draw_Line ((1.5, 4.0), (5.5, 4.0));

    -- Draw right side
    Draw_Line ((5.5, 4.0), (5.5, 0.0));

    -- Draw bottom
    Draw_Line ((5.5, 0.0), (1.5, 0.0));

    -- Draw diagonal top left to bottom right
    Draw_Line ((1.5, 4.0), (5.5, 0.0));

    -- Draw diagonal top right to bottom left
    Draw_Line ((5.5, 4.0), (1.5, 0.0));

end Draw_Flag;
```

### RATIONALE FOR THE SOLUTION

The first line of code ties procedure Draw\_Flag to package Simple\_Graphics and allows us to simply write Draw\_Line instead of Simple\_Graphics.Draw\_Line. The "with" clause gives procedure Draw\_Flag access to Simple\_Graphics while the "use" clause allows us to write the procedure name Draw\_Line without the prefix "Simple\_Graphics."

The keyword "procedure" indicates this is a procedure. It is named Draw\_Flag. The keyword "is" indicates that local data follows. Draw\_Flag does not require local data. The word "begin" precedes the statements that are executed when Draw\_Flag is called. There are six procedure calls to Draw\_Line. Each call to Draw\_Line has two actual parameters of type Point whose components are of type Float. In the first call to Draw\_Line, (1.5, 4.0) is an array of two Float components as is (1.5, -4.0).

## ALTERNATIVE SOLUTION

```
with Simple_Graphics;
procedure Draw_Flag is

    -- This procedure will draw the flag
    -- by first drawing the entire flag
    -- pole from (1.5, 4.0) to (1.5, -4.0).
    -- The top, right side, and bottom
    -- of the flag will be drawn next,
    -- in the order specified. Finally
    -- the two diagonal lines are drawn.

    -- This procedure does not label the
    -- diagram with the coordinates of the points.

begin -- Draw_Flag

    -- Draw entire flag pole
    Simple_Graphics.Draw_Line ((1.5, -4.0), (1.5, 4.0));

    -- Draw top
    Simple_Graphics.Draw_Line ((1.5, 4.0), (5.5, 4.0));

    -- Draw right side
    Simple_Graphics.Draw_Line ((5.5, 4.0), (5.5, 0.0));

    -- Draw bottom
    Simple_Graphics.Draw_Line ((5.5, 0.0), (1.5, 0.0));

    -- Draw diagonal top left to bottom right
    Simple_Graphics.Draw_Line ((1.5, 4.0), (5.5, 0.0));

    -- Draw diagonal top right to bottom left
    Simple_Graphics.Draw_Line ((5.5, 4.0), (1.5, 0.0));

end Draw_Flag;
```



## DISCUSSION

Use of the "with" clause gives Draw\_Flag access to package Simple\_Graphics. By omitting the "use" clause in the alternative solution we must access the elements of package Simple\_Graphics using dot notation. Therefore each call to Draw\_Line in Draw\_Flag as presented in the alternative solution must be written as

```
Simple_Graphics.Draw_Line (...);
```

By including the "use" clause in the solution, the prefix "Simple\_Graphics." is no longer required in each call to Draw\_Line.

## EXERCISE 2.2

### OBJECTIVE

To explain the concept of an Ada procedure.

### TUTORIAL

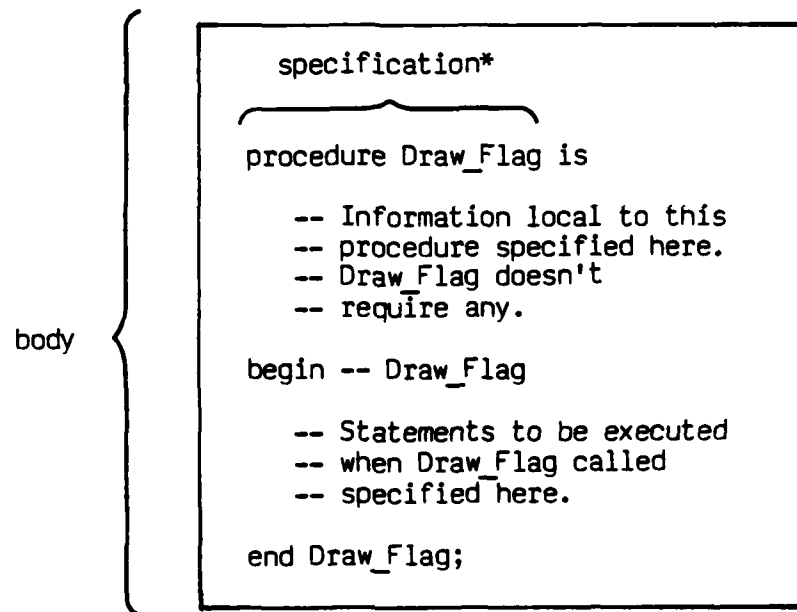
An Ada procedure is one of Ada's program units. The following shows all of Ada's program units.

- packages
- subprograms
  - procedures
  - functions
- tasks
- generic program units

Packages were introduced in Exercise 2.1. This exercise discusses procedures. Functions are addressed in Exercise 2.3. Tasks are not addressed in this workbook. The second workbook, Advanced Ada, introduces tasking. The third workbook, Real-time Ada, provides a thorough treatment of tasking and illustrates how tasking applies to real time systems. Generic program units are covered in Exercise 11.4.

A procedure is one of Ada's workhorse units. There are two parts to an Ada procedure, a specification and a body. Figure 2-2 shows the specification and body for Draw\_Flag introduced in Exercise 2.1. The specification states the procedure's interface with the caller. Specifically, a procedure specification

- states that it is a procedure
- names the procedure
- lists the formal parameters and their data types in parentheses



\*If Draw\_Flag had parameters they would be part of the specification.

Figure 2-2. Procedure Draw\_Flag Structure

The procedure body includes the procedure specification and in addition supplies the actual code which is executed when the procedure is invoked or called. Specifically, a procedure body

- states that it is a procedure
- names the procedure
- lists the formal parameters and their data types in parentheses
- lists local data required by the procedure
- contains executable code.

The format for a procedure specification is:

```
procedure Name_Of_Procedure (formal parameters)
```

The format for a procedure body is:

```
procedure Name_Of_Procedure (formal parameters) is
    -- Local data placed here. Upon leaving the procedure, the
    -- local data no longer exists.
    -- In addition to local data, local
    -- procedures (and functions which you will meet in Exercise 2.3)
    -- which would be invoked in the statements following
    -- the "begin" of Name_of_Procedure
    -- may be declared here. The problem at the end of
    -- this exercise deals with this issue

begin -- Name_Of_Procedure
    -- Statements to be executed when a
    -- Name_Of_Procedure is called placed
    -- here

end Name_Of_Procedure;
```

Parameterless procedures, e.g. Draw\_Flag, have no formal parameters, and therefore do not require parentheses.

In summary you have been exposed to the following concepts:

- a procedure declaration in a package specification
- a procedure body in a package body
- a main procedure like Draw\_Flag

Let's discuss each of these. To do this we need to revisit the package specification for Simple\_Graphics.

```
package Simple_Graphics is  
  
    type Point is array (1 .. 2) of Float;  
    procedure Draw_Line (From : in Point; To : in Point);  
    procedure Draw_Circle (Center : in Point; Radius : in Float);  
  
end Simple_Graphics;
```

There are two procedure declarations in this package. The two declarations state that these procedures are supplied by this package. The format for a procedure declaration is a procedure specification followed by a terminating semicolon.

Where the procedure specification states what the procedure provides (i.e. the interface between the procedure and its caller), the procedure body states how the procedure is implemented and is hidden in the package body. For example, the procedure specification for Draw\_Line states that it is a procedure, its name is Draw\_Line, and it requires two parameters of type Point (i.e. the two points to be connected by a line). The body for Draw\_Line is in the body of package Simple\_Graphics.

Remember, a package supplies resources. If a procedure is a resource supplied by a package the fact that it is a resource is stated in the package specification, while its implementation is found in the package body.

A procedure may appear by itself as a main procedure, as is the case with procedure Draw\_Flag introduced in Exercise 2.1.

Finally, as indicated on the previous page, procedures may be nested.

As an example of a procedure with parameters the following assumes access to Simple\_Graphics and represents a procedure to draw a square. Draw\_Square requires two parameters: the position of the top left corner of the square and the length of a side, of types Point and Float respectively. The square will always be drawn with the top and bottom sides parallel to the X-axis.

```
with Simple_Graphics; use Simple_Graphics;  
procedure Draw_Square (Top_Left : in Point; Side_Length : in Float)  
is
```

```
-- Local data declarations go here. Local data declared here  
-- cannot be accessed outside of this procedure.
```

```
-- Declare three new data items for the  
-- three remaining corners of the square  
-- to improve the readability of the code.
```

```
Top_Right    : Point := (Top_Left(1) + Side_Length,  
                          Top_Left(2));  
Bottom_Right : Point := (Top_Left(1) + Side_Length,  
                          Top_Left(2) - Side_Length);  
Bottom_Left  : Point := (Top_Left(1),  
                          Top_Left(2) - Side_Length);
```

```
begin -- Draw_Square
```

```
-- Draw top of square
```

```
Draw_Line (Top_Left, Top_Right);
```

```
-- Draw right side of square
```

```
Draw_Line (Top_Right, Bottom_Right);
```

```
-- Draw bottom of square
```

```
Draw_Line (Bottom_Right, Bottom_Left);
```

```
-- Draw left side of square
```

```
Draw_Line (Bottom_Left, Top_Left);
```

```
end Draw_Square;
```

This page intentionally left blank

PROBLEM

Rewrite procedure Draw\_Flag making use of the new procedure Draw\_Square.  
(Hint: Consider Draw\_Square as local to procedure Draw\_Flag.)



This page intentionally left blank

## SOLUTION TO EXERCISE 2.2

### SOLUTION

```

with Simple_Graphics; use Simple_Graphics;
procedure Draw_Flag is
    -- This procedure will draw the flag by
    -- first drawing the square. It will
    -- then draw the two diagonal lines,
    -- and finally draw the remainder of the flag pole.
    -- This procedure does not label the
    -- diagram with the coordinates of the points.
    procedure Draw_Square (Top_Left : in Point; Side_Length : in Float) is
        -- Local data declarations go here. Local data declared here
        -- cannot be accessed outside of this procedure.
        -- Declare three new data items for the
        -- three remaining corners of the square
        -- to improve the readability of the code.
        Top_Right      : Point := (Top_Left(1) + Side_Length, Top_Left(2));
        Bottom_Right   : Point := (Top_Left(1) + Side_Length,
                                   Top_Left(2) - Side_Length);
        Bottom_Left    : Point := (Top_Left(1), Top_Left(2) - Side_Length);
    begin -- Draw_Square
        -- Draw top of square
        Draw_Line (Top_Left, Top_Right);
        -- Draw right side of square
        Draw_Line (Top_Right, Bottom_Right);
        -- Draw bottom of square
        Draw_Line (Bottom_Right, Bottom_Left);
        -- Draw left side of square
        Draw_Line (Bottom_Left, Top_Left);
    end Draw_Square;
begin -- Draw_Flag
    -- Draw the square
    Draw_Square ((1.5, 4.0), 4.0);
    -- Draw diagonal top left to bottom right
    Draw_Line ((1.5, 4.0), (5.5, 0.0));
    -- Draw diagonal top right to bottom left
    Draw_Line ((5.5, 4.0), (1.5, 0.0));
    -- Draw remainder of flag pole
    Draw_Line ((1.5, 0.0), (1.5, -4.0));
end Draw_Flag;

```

## RATIONALE FOR THE SOLUTION

The "with" and "use" clauses, which allow access to and use of the contents of package Simple\_Graphics, appear first. They appear prior to the procedure which accesses those resources. The "with" clause precedes the "use" clause.

The procedure specification starts with the word "procedure", and is followed by the name Draw\_Flag. There are no parameters for Draw\_Flag.

The word "is" indicates that this is not a declaration. If it were a procedure declaration it would be terminated with a semicolon as it is when included in a package specification. The word "is" indicates that this is the procedure body.

The local procedure required for this procedure is Draw\_Square. It is placed prior to the "begin" of Draw\_Flag. It is a local procedure and not accessible outside Draw\_Flag. Upon leaving Draw\_Flag procedure Draw\_Square no longer exists and thus cannot be called.

The word "begin" indicates the beginning of the executable portion of the procedure. Following the word "begin" is where the statements are placed that will be executed when Draw\_Flag is invoked. The call to Draw\_Square and the three calls to Draw\_Line appear next. First the square is drawn, then the remaining three lines are drawn to complete the flag.

Finally the word "end" followed by "Draw\_Flag;" indicates the end of the procedure. Use of the procedure name following the word "end" is optional. In this workbook the name of the program unit being terminated will always be written.

### ALTERNATIVE SOLUTION

with Simple\_Graphics; use Simple\_Graphics;  
procedure Draw\_Flag is

-- This procedure will draw the flag by  
-- first drawing the square. It will  
-- then draw the two diagonal lines,  
-- and finally draw the remainder of the flag pole.

-- This procedure does not label the  
-- diagram with the coordinates of the points.

procedure Draw\_Square (Top\_Left : in Point; Side\_Length : in Float) is

-- Local data declarations go here. Local data declared here  
-- cannot be accessed outside of this procedure.

-- Declare three new data items for the  
-- three remaining corners of the square  
-- to improve the readability of the code.

Top\_Right : Point := (Top\_Left (1) + Side\_Length, Top\_Left (2));  
Bottom\_Right : Point := (Top\_Left (1) + Side\_Length,  
Top\_Left (2) - Side\_Length);  
Bottom\_Left : Point := (Top\_Left (1), Top\_Left (2) - Side\_Length);

begin -- Draw\_Square

-- Draw top of square

Draw\_Line (Top\_Left, Top\_Right);

-- Draw right side of square

Draw\_Line (Top\_Right, Bottom\_Right);

-- Draw bottom of square

Draw\_Line (Bottom\_Right, Bottom\_Left);

-- Draw left side of square

Draw\_Line (Bottom\_Left, Top\_Left);

end Draw\_Square;

```

procedure Draw_Diagonals is
    -- No local data
begin -- Draw_Diagonals
    -- Draw diagonal top left to bottom right
    Draw_Line ((1.5, 4.0), (5.5, 0.0));
    -- Draw diagonal top right to bottom left
    Draw_Line ((5.5, 4.0), (1.5, 0.0));
end Draw_Diagonal;

begin -- Draw_Flag
    -- Draw the square
    Draw_Square ((1.5, 4.0), 4.0);
    -- Draw diagonals
    Draw_Diagonals;
    -- Draw remainder of flag pole
    Draw_Line ((1.5, 0.0), (1.5, -4.0));
end Draw_Flag;

```

## DISCUSSION

The multiple procedure calls to Draw\_Line in the Solution to Exercise 1.1 are confusing. It is time consuming to determine the order in which the lines are drawn. Here, use of the procedure Draw\_Square in both the solution and alternative solution makes the code clearer to read, and therefore easier to maintain. Draw\_Square operates at a higher level of abstraction than Draw\_Line. Additionally, inclusion of a second nested procedure, Draw\_Diagonals in the alternative solution creates even clearer code. This procedure would, however, be better written with parameters indicating point coordinates, as in Draw\_Square, rather than assuming global values. The executable code in the alternative solution operates at a slightly higher level of abstraction than that of the solution.

This page intentionally left blank

## EXERCISE 2.3

### OBJECTIVE

To explain the concept of an Ada function.

### TUTORIAL

An Ada function is an Ada program unit as stated in Exercise 2.2. A function is another of Ada's workhorse units. There are two parts to an Ada function, a specification and a body. Figure 2-3 shows the structure of a function called Area Of Circle which calculates the area of a circle. The specification defines the function interface. Specifically, a function specification

- states that it is a function
- names the function
- lists the formal parameters and their data types in parentheses
- indicates the type of the value to be returned

The function body includes the function specification and in addition supplies the actual code which is executed when the function is called. Specifically a function body

- states that it is a function
- names the function
- lists the formal parameters and their data types in parentheses
- indicates the type of the value to be returned
- lists local data required by the function
- contains executable code
- explicitly returns a value

The format for a function specification is as follows:

```
function Name Of Function ( formal parameters ) return Type_Name
```



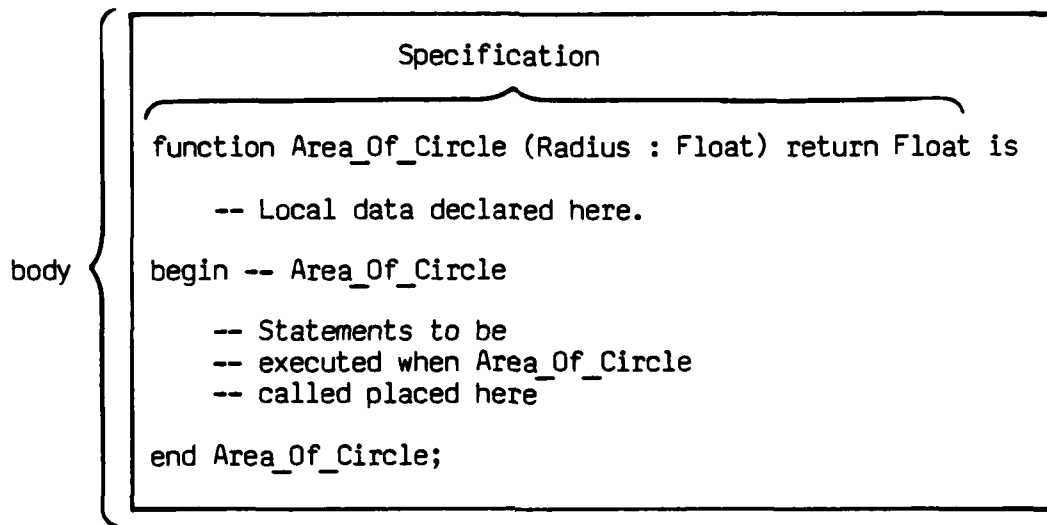


Figure 2-3. Function Area\_Of\_Circle Structure

Notice the structure of a function body. Essentially it is the same as a procedure.

```
function Function_Name ( parameter list ) return Type_Name is
    -- Local data necessary for the function
    -- to execute placed here

begin -- Function_Name

    -- Executable statements placed here. A function
    -- must include at least one return statement.

    return Some_Value;

end Function_Name;
```

The following shows the complete code for function Area\_Of\_Circle.

```
function Area_Of_Circle (Radius : Float) return Float is
    -- No local data required

begin -- Area_Of_Circle

    -- Return the area of a circle which
    -- is calculated as Pi times
    -- the radius squared

    return 3.141593 * Radius ** 2;

end Area_Of_Circle;
```

Function Area\_Of\_Circle has the required elements:

- the word "function"
- a name, Area\_Of\_Circle
- the parameter Radius and its type, Float, separated by the required colon and enclosed within parentheses
- the required word, "return," followed by the type of the value to be returned, i.e. Float.
- the word "is" indicating this is a body not a declaration

Additionally, the word "begin" precedes the executable statements of the function. For this function there is only one, the return statement. We want to return the value of the area of the circle. The formula for the area of a circle is  $\text{PI} \times R^2$  where R represents the radius of the circle. In Ada we write `3.141593 * Radius ** 2` to represent that expression. (The \* indicates multiplication, and the \*\* indicates exponentiation.) The value is calculated and returned to the point of call of the function.

Finally the word "end" directly followed by "Area\_Of\_Circle;" terminates the function body.

There is an alternative way to code function Area\_Of\_Circle using local data as follows:

```
function Area_Of_Circle (Radius : Float) return Float is
    -- Declare a data item Circle_Area to hold the value
    -- calculated for the area.
    -- Circle_Area is not accessible outside this function.
    -- Upon completion of Area_Of_Circle, Circle_Area
    -- no longer exists

    Circle_Area : Float;
begin -- Area_Of_Circle
    -- Area is calculated as Pi times the radius squared
    Circle_Area := 3.141593 * Radius ** 2;
    -- Return area of circle
    return Circle_Area;
end Area_Of_Circle;
```

A procedure call is a stand alone statement as shown in Exercises 2.1 and 2.2. How does one call a function? A function call is not a stand alone statement. A function call may only appear in an expression. Remember, a function returns a value.

The following procedure draws a circle centered at (2,0) with radius of 2, and calculates its area.

```
with Simple_Graphics; use Simple_Graphics;
with Simple_IO;       use Simple_IO;
procedure Circle is

    -- This procedure draws a circle
    -- centered at (2.0, 0.0) with radius 2.0 and calculates
    -- the area.

    The_Area : Float;

    function Area_Of_Circle (Radius : Float) return Float is

        -- No local data required

    begin -- Area_Of_Circle

        -- Return the area of a circle which
        -- is calculated as Pi times
        -- the radius squared

        return 3.141593 * Radius ** 2;

    end Area_Of_Circle;

begin -- Circle

    -- Draw Circle

    Draw_Circle ((2.0, 0.0), 2.0);

    -- Compute area. The function call
    -- Area_Of_Circle (2.0)
    -- appears in an expression.

    The_Area := Area_Of_Circle (2.0);
    Put (The_Area);

end Circle;
```

As another example, consider procedure Square Root of Section 1. Two statements of this procedure are of interest now. They are

```
Get (X);  
Put (Sqrt (X));
```

The first statement, "Get (X);" is a procedure call. The statement invokes the procedure Get with the parameter X. The second statement, "Put (Sqrt (X));" is also a procedure call. It invokes the procedure named Put with the actual parameter whose value is the result of evaluating the expression Sqrt (X). Sqrt (X) is an expression whose value is returned by the function named Sqrt with the parameter X.

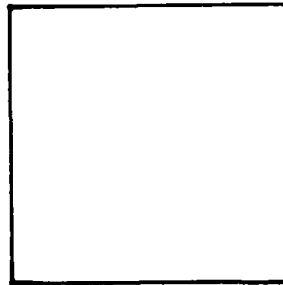
Notice the differences between a function and a procedure. They are

- a procedure call is a statement whereas a function call is included as part of an expression
- the word "return" appears in a function specification but does not appear in a procedure specification
- a function returns only one value whose type is indicated after the word return in the function specification
- a procedure can communicate many (or no) results through parameters (see Exercise 2.4)
- the return statement in a function body returns the value of the function

### PROBLEM

Write a function called `Area_Of_Square` which returns the area of a square given the length of a side. Write a main procedure called `Display_Square` which draws the following square and calculates its area.

(1.0, 1.0)



(3.0, 1.0)

This page intentionally left blank

## SOLUTION TO EXERCISE 2.3

### SOLUTION

```
with Simple_Graphics; use Simple_Graphics;
with Simple_IO;       use Simple_IO;
procedure Display_Square is

  -- Local procedure to draw square.
  -- Local variable Square_Area to hold area.

  Square_Area : Float;

  function Area_Of_Square (Side_Length : Float) return Float is

    -- No local data required

  begin -- Area_Of_Square

    -- Area is side squared

    return Side_Length ** 2;

  end Area_Of_Square;
```



```

procedure Draw_Square (Top_Left : in Point; Side_Length : in Float) is
    -- Local data declarations go here. Local data declared here
    -- cannot be accessed outside of this procedure.

    -- Declare three new data items for the
    -- three remaining corners of the square
    -- to improve the readability of the code.

    Top_Right    : Point := (Top_Left(1) + Side_Length, Top_Left(2));
    Bottom_Right : Point := (Top_Left(1) + Side_Length,
                             Top_Left(2) - Side_Length);
    Bottom_Left  : Point := (Top_Left(1), Top_Left(2) - Side_Length);

begin -- Draw_Square

    -- Draw top of square

    Draw_Line (Top_Left, Top_Right);

    -- Draw right side of square

    Draw_Line (Top_Right, Bottom_Right);

    -- Draw bottom of square

    Draw_Line (Bottom_Right, Bottom_Left);

    -- Draw left side of square

    Draw_Line (Bottom_Left, Top_Left);

end Draw_Square;

begin -- Display_Square

    Draw_Square ((1.0, 1.0), 2.0);           -- procedure call
    Square_Area := Area_Of_Square (2.0);    -- function call
    Put (Square_Area);                       -- procedure call

end Display_Square;

```

## RATIONALE FOR THE SOLUTION

The function `Area_Of_Square` has the required word "function," the appropriate user defined name `Area_Of_Square` with its parameter `Side_Length` enclosed in parentheses and separated from its type by the required colon. The required word "return" is followed by the type of the value to be returned by the function, in this case `Float`. Finally the word "is" indicates that this is not a specification or declaration but the body. There is no requirement for local data. The "begin" indicates the start of the executable code which consists of one return statement followed by the expression to be calculated and returned. This expression is of type `Float`. The word "end" optionally followed by the function name `Area_Of_Square` terminates the function.

The procedure `Display_Square` has a local subprogram, the procedure `Draw_Square` which we met in Exercise 2.2. The "begin" introduces the executable code of the procedure which here consists of just two statements: one to draw the square, the other to compute the area. As stated as a comment in the code, we will learn how to output this value later.

## ALTERNATIVE SOLUTION

```
function Area_Of_Square (Side_Length : Float) return Float is
    The_Area : Float;
begin -- Area_Of_Square
    -- Area is side length multiplied by
    -- itself

    The_Area := Side_Length * Side_Length;
    return The_Area;
end Area_Of_Square;
```

## DISCUSSION

The alternative solution introduces a local data item called `The_Area` and additionally computes the area of the square by using the multiplication operator rather than exponentiation. Use of the data item `The_Area` makes the return statement more readable at the expense of additional variables being introduced.

This page intentionally left blank

## EXERCISE 2.4

### OBJECTIVE

To illustrate Ada's parameter modes.

### TUTORIAL

Up to this point you have seen only one kind of parameter. All the parameters that you were exposed to in the first 3 exercises are called "in" parameters. Recall the specification for package Simple\_Graphics:

```
package Simple_Graphics is
    type Point is array (1 .. 2) of Float;
    procedure Draw_Line (From : in Point;
                        To   : in Point);
    procedure Draw_Circle (Center : in Point;
                          Radius  : in Float);
end Simple_Graphics;
```

Notice the word "in" prior to the name of the type in each of the procedure declarations. This specifies that the parameters can only be read. An "in" parameter acts as a local constant and therefore may not be modified within its enclosing program unit. In procedure Draw\_Line, the parameters From and To cannot be modified.

Ada provides three parameter modes: "in," "in out," and "out." (See Figure 2-4). The mode, if specified, must be specified prior to the type name of the parameter. If the parameter mode is not specified, the parameter mode is assumed to be "in," i.e. the default mode for parameters is "in."

Let us revisit function Area\_Of\_Circle introduced in Exercise 2.3.

<u>in</u>	<u>out</u>	<u>in out</u>
local constant	local variable	local variable
value provided by actual parameter	value assigned to actual parameter during execution of subprogram.	value provided by actual parameter and assigned to actual parameter during execution of subprogram.
cannot be modified by procedure or function	cannot be read by the procedure	may be read and modified by procedure

Figure 2-4. Parameter Modes

```

function Area_Of_Circle (Radius : Float) return Float is
    -- No local data required
begin -- Area_Of_Circle
    -- Area is calculated as pi times
    -- the radius squared
    return 3.141593 * Radius ** 2;
end Area_Of_Circle;

```

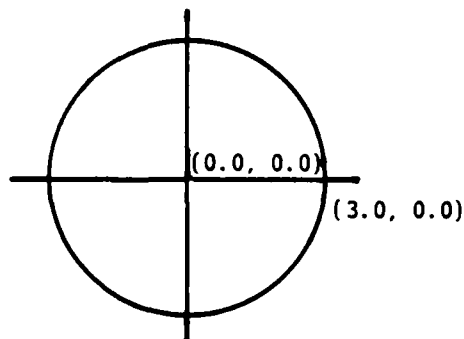
Upon execution of the return statement the function returns the value of the area to the caller. Another approach is to code Area\_Of\_Circle as a procedure as follows:

```

procedure Area_Of_Circle (Radius      : in Float;
                          Circle_Area : out Float) is
    -- No local data required
begin
    Circle_Area := 3.141593 * Radius ** 2;
end Area_Of_Circle;

```

The procedure specification indicates that there are two parameters. The first parameter is Radius of type Float which is an "in" parameter (by default) and cannot be modified. It is used in calculating the area of a circle. The second parameter, Circle\_Area, is an "out" parameter and supplies a value to the actual parameter of the caller. Let's examine a procedure to draw the following circle and compute the area.



The code is as follows:

```

with Simple_Graphics; use Simple_graphics; -- makes Draw_Circle
with Simple_IO;       use Simple_IO;       --      available
procedure Display_Circle is

    -- Local procedures to draw the square
    -- and find the area.

    -- A local variable to hold the area.

    Area : Float; -- to hold value of area

    procedure Area_Of_Circle (Radius      : in Float;
                               Circle_Area : out Float) is

        -- No local data required

    begin -- Area_Of_Circle

        Circle_Area := 3.1415923 * Radius ** 2;

    end Area_Of_Circle;

begin -- Display_Circle

    Draw_Circle ((0.0, 0.0), 3.0); -- Draw the circle
    Area_Of_Circle (3.0, Area);    -- Get area of circle
    Put (Area);                   -- Print area of circle

end Display_Circle;

```

An "out" parameter acts as a variable, local to the calling program unit, whose value is assigned as a result of the execution of the procedure. Here the value for the area of the circle is returned to the variable Area upon completion of the procedure call

```
Area_Of_Circle (3.0, Area);
```

Consider the following procedure to interchange two values.

```
procedure Interchange (First, Second : in out Float) is
    Temporary : Float;
begin -- Interchange
    Temporary := First;    -- save First
    First     := Second;   -- Replace First with Second
    Second    := Temporary; -- Replace Second with saved value
end Interchange;
```

This procedure interchanges the two values passed to it. After the call, the values passed to First and Second are reversed. Here Interchange needs to read and modify the values before returning them to the caller. An "in out" parameter does act as a local constant in that the value is provided by the actual parameter, but in the sense that it can be modified as a result of execution of the procedure, it really acts as a local variable. (See Figure 2-4).

In summary, the three parameter modes may be thought of in terms of data flow. Parameters of mode "in" can be thought of as flowing into the program unit being called while parameters of mode "out" flow out of the called program unit. Those parameters of type "in out" flow in both directions. See Figure 2-5.



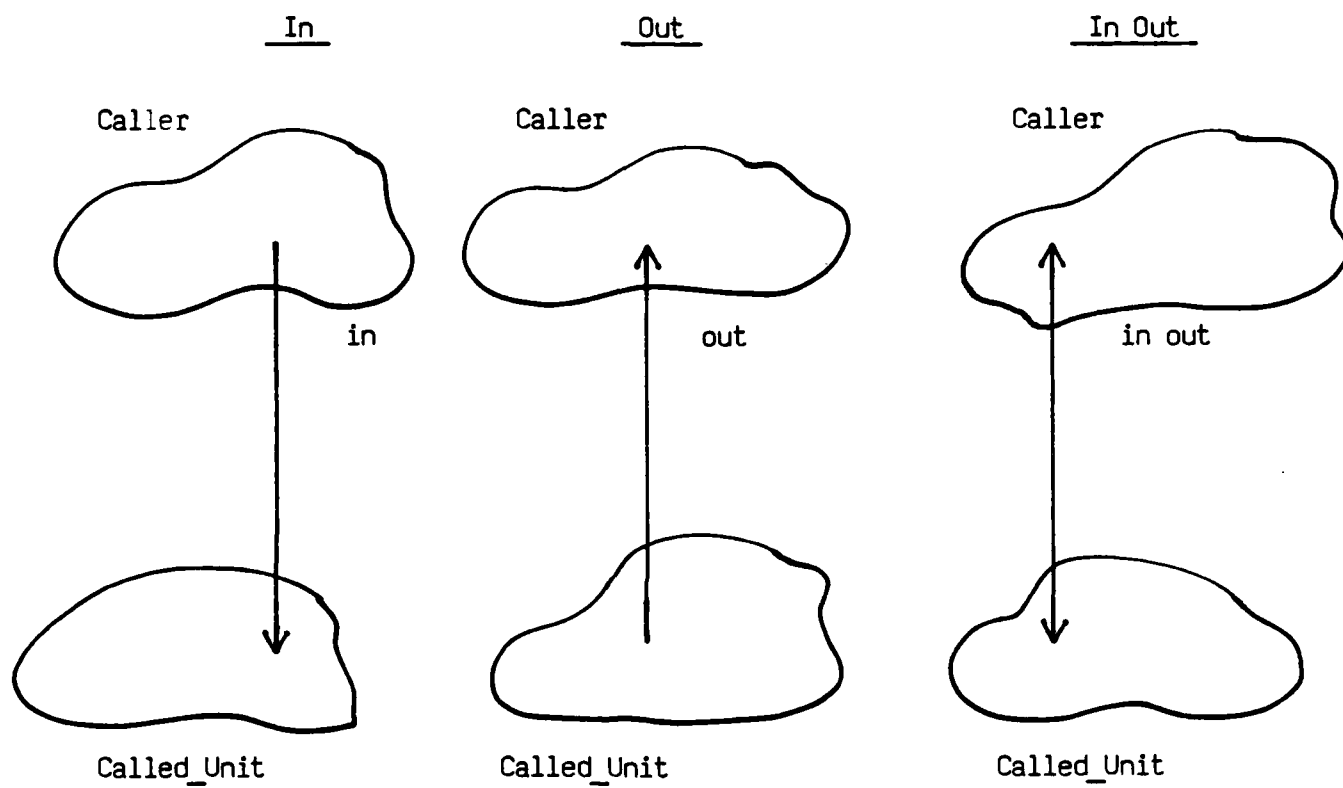
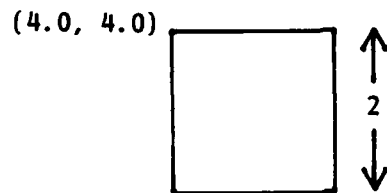


Figure 2-5. Parameter Modes as Data Flow

PROBLEM

Write a subprogram to calculate the area of a square given the length of a side. Write a procedure called Display\_Square which draws the following square and calculates its area.



This page intentionally left blank

## SOLUTION TO EXERCISE 2.4

### SOLUTION

```
with Simple_Graphics; use Simple_Graphics;
with Simple_IO;       use Simple_IO;
procedure Display_Square is

    -- Local procedures to draw the square and
    -- calculate the area.

    -- Local data item required to hold the area.

    The_Area : Float;

    procedure Area_Of_Square (Side_Length : in Float;
                              Area        : out Float) is

        -- No local data item Area required

    begin -- Area_Of_Square

        Area := Side_Length ** 2;

    end Area_Of_Square;

    procedure Draw_Square (Top_Left      : in Point;
                           Length_Of_Side : in Float) is

        -- You have seen this before (Exercise 2.2, 2.3)

    end Draw_Square;

begin -- Display_Square

    Draw_Square ((4.0, 4.0), 2.0);    -- Draw the square
    Area_Of_Square (2.0, The_Area);   -- Calculate the Area
    Put (The_Area);                  -- Output Result

end Display_Square;
```

#### RATIONALE FOR THE SOLUTION

The instructions explicitly state to find the area of the square. It was not specified whether to code it as a function or a procedure. The solution chose to code finding the area as a procedure. The only way for the procedure to supply the value representing the area to a caller is by a parameter. The value is calculated internally in procedure Area\_Of\_Square. It is not supplied by the calling subprogram and then modified. Therefore it must be an "out" rather than an "in" or "in out" parameter.

#### ALTERNATIVE SOLUTION

Area\_Of\_Square might have been implemented as a function as follows:

```

with Simple_Graphics; use Simple_Graphics;
with Simple_IO;       use Simple_IO;
procedure Display_Square is

    -- Local function to calculate the area
    -- Local procedure to draw the square

    -- Local data item required to hold the area.
    The_Area : Float;

    function Area_Of_Square (Side_Length : Float) return Float is

        -- No local data item required

    begin -- Area_Of_Square

        return Side_Length ** 2;

    end Area_Of_Square;

    procedure Draw_Square (Top_Left      : in Point;
                           Length_Of_Side : in Float) is

        -- You have seen this before

    end Draw_Square;

begin -- Display_Square

    Draw_Square ((4.0, 4.0), 2.0);    -- Draw the square
    The_Area := Area_Of_Square (2.0); -- Calculate the Area
    Put (The_Area);                  -- Output Result

end Display_Square;

```

### DISCUSSION

Since the purpose of Area\_Of\_Square is to compute a single value, a function seems more appropriate.

This page intentionally left blank

## EXERCISE 2.5

### OBJECTIVE

To illustrate positional and named notation for specification of actual parameters.

### TUTORIAL

Ada provides alternative notations for specifying actual parameters. They are called positional notation and named notation.

With positional notation the first actual parameter is substituted for the first formal parameter, the second actual parameter is substituted for the second formal parameter, and so on. For example the procedure call

Interchange (5.0, 8.0);

specifies two actual parameters, 5.0 and 8.0, to be passed to the formal parameters First and Second of procedure Interchange found in Exercise 2.4. The first actual parameter 5.0 is substituted for the formal parameter, First. The second actual parameter is substituted for the formal parameter, Second. Positional notation uses the order of appearance of the actual parameters within the parameter list as the mechanism for relating actual parameters to formal parameters.

Named notation uses the name of the formal parameter to explicitly state which formal parameter the actual parameter is to be associated with. For example the above call in named notation is either

Interchange (First => 5.0, Second => 8.0);

or

Interchange (Second => 8.0, First => 5.0);

Notice that in named notation the physical ordering of the actual parameters is immaterial.

Named notation and positional notation may be mixed in one call; however, all positional parameters must come first and in their correct order.



This page intentionally left blank

PROBLEM

Recode procedure Display\_Square of Exercise 2.4 using named notation.

This page intentionally left blank

## SOLUTION TO EXERCISE 2.5

### SOLUTION

```
with Simple_Graphics; use Simple_Graphics;
with Simple_IO;       use Simple_IO;
procedure Display_Square is

    -- Local procedures to draw the square and
    -- calculate the area.

    -- Local data item required to hold the area.

    The_Area : Float;
    procedure Area_Of_Square (Side_Length : in Float;
                              Area        : out Float) is

        -- No local data item Area required

    begin -- Area_Of_Square

        Area := Side_Length ** 2;

    end Area_Of_Square;

    procedure Draw_Square (Top_Left      : in Point;
                           Length_Of_Side : in Float) is

        -- You have seen this before (Exercise 2.2)

    end Draw_Square;

begin -- Display_Square

    Draw_Square (Top_Left      => (4.0, 4.0),
                 Length_Of_Side => 2.0);
    Area_Of_Square (Side_Length => 2.0,
                   Area         => The_Area);
    Put (The_Area);

end Display_Square;
```

## RATIONALE FOR THE SOLUTION

In both procedure calls, all actual parameters are specified in named notation. The name of the formal parameter is specified, followed by an arrow ( => ), then directly followed by the actual parameter.

## ALTERNATIVE SOLUTION

```
with Simple_Graphics; use Simple_Graphics;
with Simple_IO;       use Simple_IO;
procedure Display_Square is

    -- Local procedures to draw the square and
    -- calculate the area.

    -- Local data item required to hold the area.

    The_Area : Float;

    procedure Area_Of_Square (Side_Length : in Float;
                              Area         : out Float) is

        -- No local data item Area required

    begin -- Area_Of_Square

        Area := Side_Length ** 2;

    end Area_Of_Square;

    procedure Draw_Square (Top_Left      : in Point;
                           Length_Of_Side : in Float) is

        -- You have seen this before

    end Draw_Square;

begin -- Display_Square

    Draw_Square (Length_of_Side => 2.0,
                 Top_Left       => (4.0, 4.0) );
    Area_Of_Square (Area        => The_Area,
                   Side_Length  => 2.0);
    Put (The_Area);

end Display_Square;
```

## DISCUSSION

The difference between the two solutions is the order of the parameters in the calls to `Draw_Square` and `Area_Of_Square`.

Both the solution and the alternative solution used named notation for specifying all the actual parameters. Named notation is convenient, for instance, where conventions differ. For example dates in America are specified as month, day, year while dates in Europe are specified by stating the day first, followed by the month and then followed by the year. A procedure specification for calculating an integer representation for the day of the year follows:

```
procedure Calculate_Day_Of_Year (Month      : in Month_Type;
                                Day         : in Day_Type;
                                Year        : in Year_Type;
                                Day_Of_Year : out Day_Of_Year_Type);
```

In America, `Calculate_Day_Of_Year` will most likely be called using positional notation as follows:

```
Calculate_Day_of_Year (Mar, 7, 1983, Number_Of_Day);
```

This call orders the actual parameters in a way unnatural to most Europeans. A European may code the call using named notation as follows:

```
Calculate_Day_Of_Year (Day      => 7,
                      Month     => Mar,
                      Year      => 1983,
                      Day_Of_Year => Number_Of_Day);
```

Use of named notation here enables him to use the same procedure, yet express the parameters in a way consistent with his day to day notation.

This page intentionally left blank

## EXERCISE 2.6

### OBJECTIVE

To introduce package bodies and the concept and structure of an Ada program.

### TUTORIAL

As previously stated, Ada provides the following four program units:

- packages
- subprograms
  - procedures
  - functions
- tasks
- generic program units

Exercise 2.1 illustrated that a package houses logically related items. Exercise 2.2 introduced procedures as one of the workhorse units. The second workhorse unit is the Ada function which was discussed in Exercise 2.3. Tasks are not addressed in the Ada Primer. They are addressed in detail in Real-Time Ada. Generics are introduced in Exercise 11.4.

Each of the four program units has two parts: a specification and a body. In the case of a package, the specification and body are physically separate.

A package specification has the following form:

```
package Package_Name is
    -- Declarations
end Package_Name;
```

as in the specification for package Simple\_Graphics introduced in Exercise 2.1.



A package body has the following structure:

```
package body Package_Name is
    -- declarations
begin -- Package_Name
    -- Initialization statements
end Package_Name;
```

Within the package body appear the implementations of the items declared in the corresponding package specification. Now it is time to examine the structure of the package body for Simple\_Graphics. Its structure is as follows:

```
package body Simple_Graphics is
    -- The procedure body for Draw_Line
    procedure Draw_Line (From : in Point; To : in Point) is
        -- Local data declarations
    begin -- Draw_Line
        -- The code which draws the line
    end Draw_Line;

    procedure Draw_Circle (Center : in Point;
                           Radius : in Float) is
        -- Local data declarations
    begin -- Draw_Circle
        -- Code for drawing the circle here
    end Draw_Circle;
end Simple_Graphics;
```

Assuming we have the code for the two procedures we now have a package specification and a package body. Assume these two units have been compiled into a program library. The Ada language requires that Ada compilers allow submission of and enforce the language rules for Ada programs submitted as a single compilation or several compilations of physically separate units.

Next, let us revisit our old friend Draw\_Flag. It is a main procedure which needs the resources provided by Simple\_Graphics to draw a specific flag.

```
with Simple_Graphics; use Simple_Graphics;
procedure Draw_Flag is
```

```
-- This procedure will draw the flag by
-- first drawing the square. It will
-- then draw the two diagonal lines,
-- and finally draw the remainder of the flag pole.
```

```
-- This procedure does not label the
-- diagram with the coordinates of the points.
```

```
procedure Draw_Square (Top_Left : in Point; Side_Length : in Float) is
```

```
-- Local data declarations go here. Local data declared here
-- cannot be accessed outside of this procedure.
```

```
-- Declare three new data items for the
-- three remaining corners of the square
-- to improve the readability of the code.
```

```
Top_Right    : Point := (Top_Left(1) + Side_Length, Top_Left(2));
Bottom_Right : Point := (Top_Left(1) + Side_Length,
                        Top_Left(2) - Side_Length);
Bottom_Left  : Point := (Top_Left(1), Top_Left(2) - Side_Length);
```

```
begin -- Draw_Square
```

```
-- Draw top of square
```

```
Draw_Line (Top_Left, Top_Right);
```

```
-- Draw right side of square
```

```
Draw_Line (Top_Right, Bottom_Right);
```

```
-- Draw bottom
```

```
Draw_Line (Bottom_Right, Bottom_Left);
```

```
-- Draw left side of square
```

```
Draw_Line (Bottom_Left, Top_Left);
```

```
end Draw_Square;
```

```
begin -- Draw_Flag
    -- Draw the square
    Draw_Square ((1.5, 4.0), (4.0));
    -- Draw diagonal top left to bottom right
    Draw_Line ((1.5, 4.0), (5.5, 0.0));
    -- Draw diagonal top right to bottom left
    Draw_Line ((5.5, 4.0), (1.5, 0.0));
    -- Draw remainder of flag pole
    Draw_Line ((1.5, 0.0), (1.5, -4.0));
end Draw_Flag;
```

Compile this procedure into the same program library as Simple\_Graphics. Since it depends on Simple\_Graphics it cannot be compiled into the program library until the specification for Simple\_Graphics has been compiled. This leads to a discussion of the order of compilation of program units. The order in which units are encountered by the compiler is important. A unit depending on another unit cannot be compiled into a program library until all units on which it depends have been successfully compiled. The following summarizes the compilation possibilities for the package Simple\_Graphics and the procedure Draw\_Flag which depends on Simple\_Graphics.

Compilation Possibility 1

Unit 1 : Simple\_Graphics specification and body  
Unit 2 : Draw\_Flag

Compilation Possibility 2

Unit 1 : Simple\_Graphics specification  
Unit 2 : Simple\_Graphics body  
Unit 3 : Draw\_Flag

Compilation Possibility 3

Unit 1 : Simple\_Graphics specification  
Unit 2 : Draw\_Flag  
Unit 3 : Simple\_Graphics body

Compilation Possibility 4

Unit 1 : Simple\_Graphics specification  
Unit 2 : Simple\_Graphics body and Draw\_Flag

Compilation Possibility 5

Unit 1 : Simple\_Graphics specification and Draw\_Flag  
Unit 2 : Simple\_Graphics body

Compilation Possibility 6

Unit 1 : Simple\_Graphics specification  
Unit 2 : Draw\_Flag and Simple\_Graphics body

Compilation Possibility 7

Unit 1 : Simple\_Graphics specification and body  
          and Draw\_Flag

The third compilation possibility is especially interesting. It shows that Draw\_Flag can be coded and compiled prior to the coding and compilation of the package body for Simple\_Graphics. It is possible to check Draw\_Flag for syntax errors and verify its interface to package Simple\_Graphics.

There are two constant factors in each of these possibilities. The first is that the specification for Simple\_Graphics must be compiled before Draw\_Flag. The "with" and "use" clauses prior to Draw\_Flag indicate a dependency on Simple\_Graphics. Since only the specification of a package provides the interface to the outside world (i.e., the package body is not known to other program units) the only Draw\_Flag dependency is on the package specification. Draw\_Flag is not dependent on the package body. The second constant factor is that the package specification for Simple\_Graphics must be compiled before the package body.

It is time to get a little more formal. A compilation consists of one or more compilation units. A compilation unit is any unit that can be compiled separately into a program library. The following are legal Ada compilation units:

- package specification
- package body
- procedure specification
- procedure body
- function specification
- function body
- subunit

Subunits are formally addressed in Section 11; however the intuitive notion of a subunit is presented here.

For example, assume Draw\_Line has not yet been coded and we do not want to code it now. How can we allow compilation of Simple\_Graphics and any subprograms which may call Draw\_Line (e.g. Draw\_Flag) in order to partially check our code? It is done as follows:

```
package Simple_Graphics is
    -- The declarations for Float, Draw_Line, and
    -- Draw_Circle
    -- remain unchanged from Exercise 1.1
end Simple_Graphics;

package body Simple_Graphics is
    -- The procedure body for Draw_Line is
    -- replaced by a stub.

    procedure Draw_Line (From : in Point; To : in Point) is separate;

    -- The procedure body for Draw_Circle

    procedure Draw_Circle (Center : in Point; Radius : in Float) is
        -- Local declarations

    begin -- Draw_Circle

        -- Code for drawing circle here

    end Draw_Circle;
end Simple_Graphics;
```

The line "procedure Draw\_Line (From : Point; To : Point) is separate;" states that the body of Draw\_Line is not physically located in the body for Simple\_Graphics but is separate. This declaration is called a stub. Procedure Draw\_Line will appear as a physically separate unit. At a later date, the code for Draw\_Line can be compiled as a separate submission. However, it is important to realize that the unit containing the stub (e.g. package body Simple\_Graphics) must be compiled prior to the compilation of this separate unit itself (e.g. the body of Draw\_Line). It will appear as follows and is called a subunit.

```
separate (Simple_Graphics)
procedure Draw_Line (From : in Point; To : in Point) is

    -- local declarations here

begin -- Draw_Line

    -- executable statements here

end Draw_Line;
```

Subunits belong to the class of compilation units known as secondary units. Other secondary units are package bodies or, in some cases, subprogram bodies. A compilation unit that is not a secondary unit is a library unit. Why is a new term library unit introduced? The answer is that only library units may be named in a "with" clause. The "with" clause was introduced in Exercise 1.1.

The "with" clause

- denotes dependency on a library unit
- makes the name of the library unit known (accessible)



This page intentionally left blank

### PROBLEM

The following four program units are to be compiled into a program library:

Unit 1 :   package Y is  
              -- some declarations  
          end Y;

Unit 2 :   package body Y is  
              -- some stuff  
          end Y;

Unit 3 :   separate (Main)  
          procedure X is  
              -- declarations  
          begin -- X  
              -- some stuff  
          end X;

Unit 4 :   with Y; use Y;  
          procedure Main is  
              -- declarations for Main  
              procedure X is separate;  
          begin -- Main  
              -- some stuff  
          end Main;

Answer the following questions:

Q1: Which unit(s) are compilation units?

Q2: Which unit(s) are library units?

Q3: Which unit(s) are subunits?

Q4: Assuming the 4 units are to be submitted as four separate compilations in what order(s) may they be compiled?

This page intentionally left blank

## SOLUTION TO EXERCISE 2.6

### SOLUTION

#### Answer to Q1

All four units are compilation units.

#### Answer to Q2

Units 1 and 4 are Library Units.

#### Answer to Q3

Unit 3 is the only subunit.

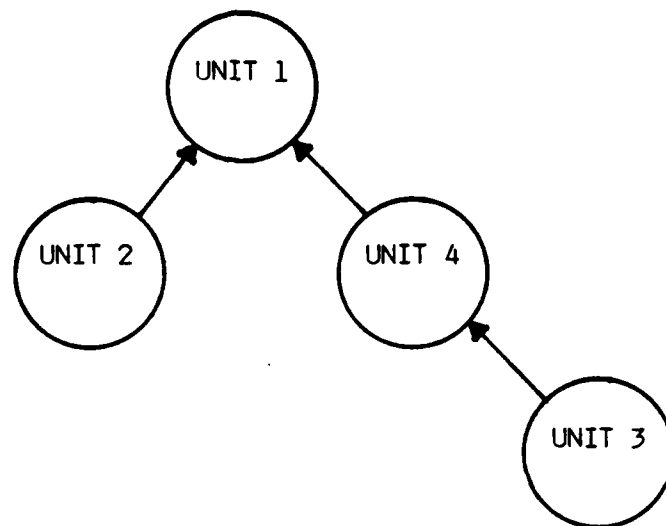
#### Answer to Q4

The following order is legal: Units 1 and 2 in one compilation, followed by Unit 4, and last but not least, Unit 3 is compiled. Other legal orders are given in the alternative solution.

### RATIONALE FOR SOLUTION

- Q1: Unit 1 is a package specification. Unit 2 is a package body. Unit 3 is a subunit of Unit 4. Unit 4 is a procedure body. All of these units are legal compilation units.
- Q2: The only compilation units that are not library units are subunits and package or subprogram bodies. Therefore Unit 3, a subunit, is not a library unit, nor is unit 2, a package body. Units 1 and 4 are compilation units which are not subunits and therefore are library units.
- Q3: A subunit is a unit which has been declared as separate. Unit 3 was declared as separate in Unit 4. Unit 3 begins with the required "separate (Parent\_Name)". Unit 3 is the only subunit.
- Q4: The package specification and package body may be compiled together into a program library. The package specification must physically precede the package body in the text file. Unit 4 is compiled next, followed by Unit 3. The parent unit, Unit 4, must be compiled before any of its children (subunits) e.g. Unit 3.

The following diagram graphically depicts the compilation requirements of the four units.



#### ALTERNATIVE SOLUTION

The only alternative solution is for Q4. As mentioned earlier, the order of compilation is a function of dependent units. The three dependencies in this group of four units are 1) Unit 4 which is dependent on Unit 1, and 2) Unit 2 which is dependent on Unit 1 and 3) Unit 3 which is dependent on Unit 4. Therefore the following possibilities exist for compiling these units into a program library:

1,2,4,3  
1,4,2,3  
1,4,3,2

#### DISCUSSION

Q4: Notice in each of these orders, Unit 1 is always compiled before Unit 4, Unit 1 is always compiled before Unit 2, and Unit 4 is always compiled before Unit 3. Those three requirements need to be satisfied. Whether Unit 3 is compiled before or after Unit 2 is immaterial.

## **Section 3**

### **LEXICAL ELEMENTS**

#### **3.1 Lexical Style**

#### **3.2 Identifiers**

This page intentionally left blank

## EXERCISE 3.1

### OBJECTIVE

To illustrate the lexical style of an Ada program.

### TUTORIAL

Section 1 introduced the small Ada program shown below which reads, calculates and prints the square roots of numbers. At that time we saw that Ada is a free format language allowing liberal use of spaces and blank lines for readability.

```
with Math_Lib; use Math_Lib;
with Simple_IO; use Simple_IO;
procedure Square_Root is

    X : Float;

begin -- Square_Root

    while not End_Of_File
    loop
        Get (X);
        Put (Sqrt (X));
    end loop;

end Square_Root;
```

Let's look at the code again from the view point of lexical style. An Ada program uses a subset of the ASCII character set (see Table 3-1). Both uppercase and lowercase letters are permitted. Combinations of these characters are used to form lexical elements. A lexical element is the smallest meaningful unit of Ada text. Table 3-2 lists the categories of legal lexical elements of an Ada program.

Let's discuss Table 3-2. An identifier is a name given to a subprogram (e.g. Main), package (e.g. Simple\_Graphics), variable (e.g. Number), as well as other Ada entities. The specific rules for writing an Ada identifier are addressed in Exercise 3.2. Identifiers include Ada reserved words. Table 3-3 lists the reserved words of the Ada language. A reserved word, though a legal Ada identifier, is predefined by the language, and as such

- may not be used by the programmer as a user defined identifier.
- may only appear in a context defined by the language.



TABLE 3-1  
ASCII CHARACTER SET

- Uppercase Letters  
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
- Digits  
0 1 2 3 4 5 6 7 8 9
- Special Characters  
" # ' ( ) \* + , - . / : ; < = > \_ | &
- The space character
- Lowercase Letters  
a b c d e f g h i j k l m n o p q r s t u v w x y z
- Special Characters - not part of Ada  
! \$ % & ' ( ) \* + , - . / : ; < = > \_ { } ~ ^ &

TABLE 3-2  
LEXICAL ELEMENTS

- Identifiers (including reserved words)
- Numeric Literals
- String Literals
- Comments
- Delimiters
  - Simple  
    & ' ( ) \* + , - . / : ; < = > |
  - Compound  
    => .. \*\* := /= >= <= << >> <>

TABLE 3-3  
ADA RESERVED WORDS

abort	declare	generic	of	select
abs	delay	goto	or	separate
accept	delta		others	subtype
access	digits	if	out	
all	do	in		task
and		is	package	terminate
array			pragma	then
at	else	limited	private	type
	elsif	loop	procedure	
	end		raise	use
begin	entry		range	
body	exception	mod	record	when
	exit		rem	while
		new	renames	with
case	for	not	return	
constant	function	null	reverse	xor

It is customary (but not required) to write reserved words in lower case. Additionally, where possible, reserved words may be shown in bold face to highlight program structure.

Numeric literals represent numeric values. Numeric literals are addressed in Exercises 4.2 and 4.4. String literals represent sequences of characters. Strings are addressed in Exercise 8.3. A comment starts with two hyphens, "--", and ends at the end of a line. Delimiters are combinations of selected special characters. See Table 3-2.

A lexical element must fit on one line. Spaces are optional between most lexical elements but are mandatory between two lexical elements which, without the separating space, could be construed as one lexical element. For example, a space is required between the two identifiers "procedure" and "Main". Without the space "procedureMain" would be a single identifier.

There are no continuation marks in Ada. Statements may freely cross line boundaries and continue on the next line. One line may contain more than one statement. Whichever the case, each statement must be terminated by a semicolon. In compound statements (i.e., statements which enclose other statements) like "if," "case," or "loop," the terminating semicolon for the compound statement appears at the end of the compound statement (e.g., after "end if" or "end loop").

As shown below, Ada programs can and should utilize indentation and blank lines to highlight program structure and enhance readability.

```
with Math_Lib; use Math_Lib;
with Simple_IO; use Simple_IO;
procedure Square_Root is

    X : Float;

begin -- Square_Root

    while not End_Of_File
    loop
        Get (X);
        Put (Sqrt (X));
    end loop;

end Square_Root;
```

This page intentionally left blank

### PROBLEM

Determine the lexical errors in the following Ada source code.

```
with Math_Lib; use Math_Lib;
with Simple_IO; use Simple_IO;
procedure New is

    --
    -- This procedure calculates the
    -- square root of a number obtained
    -- from an external device.
    --

    Number : Float;

begin

    whilenot End_Of_File
        loop
            gEt
            (Number);
            Put
            (Sqrt (NUMBER))
            end
            loop;

end;
```

AD-A165 345

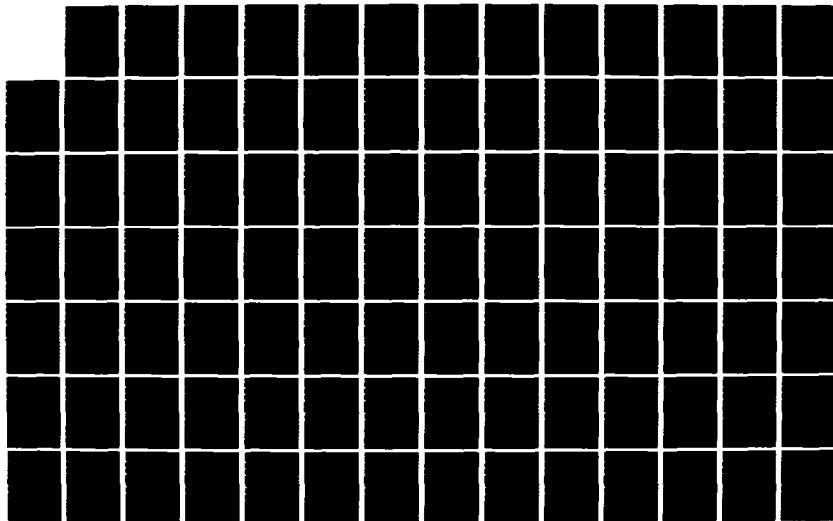
ADA (TRADEMARK) PRIMER(U) SOFTECH INC WALTHAM MA 1986  
DAB07-83-C-K506

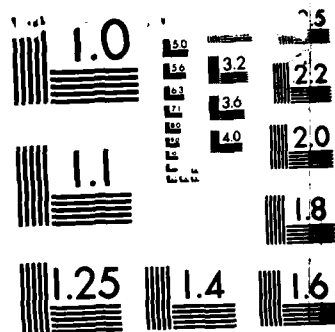
2/5

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A



This page intentionally left blank

## SOLUTION TO EXERCISE 3.1

### SOLUTION

The lines are numbered to facilitate discussion. There are 4 errors which appear in lines 3, 6, 10 and 15 respectively.

```
1  with Math_Lib ; use Math_Lib ;
2  with Simple_IO; use Simple_IO;
3  procedure New is
4
5      -- This procedure calculates the
6      square root of a number obtained
7      -- from an external device.
8
9      Number : Float;
10
11     begin
12         whilenot End_Of_File
13             loop
14                 get
15                 (Number);
16                 Put
17                 (Sqrt (NUMBER))
18                 end
19             loop;
20
21     end;
```

### RATIONALE FOR THE SOLUTION

#### Line 3

A reserved word may not be used as a user defined identifier. The word "New" is a reserved word and may only appear in the context defined by the language.

#### Line 6

Comments are a lexical element and therefore may not cross line boundaries. Two hyphens are required before the word "square".

#### Line 10

A space is needed between the identifiers "while" and "not". Without the space, the words "while" and "not" are not recognized as reserved words and "whilenot" becomes a user defined identifier.

#### Line 15

The Put statement requires the terminating semicolon.

#### ALTERNATIVE SOLUTION

There is no alternative solution to this exercise.

#### DISCUSSION

Line 9 is legal; however, the readability of the code is enhanced by indicating the name of the procedure in a comment as was done in Section 2, for example

```
begin -- New
```

There is no conflict with the use of the word "New" here as it is used within a comment.

Although somewhat strange in appearance, Line 12 is perfectly legal. In Ada no distinction is made between upper case and lower case letters.

Lines 12 and 13 combined represent the single Get statement. Statements may cross line boundaries while lexical elements (such as a comment or an identifier) may not. Similarly lines 14 and 15 are legal (once the terminating semicolon is added to line 15).

An additional comment is appropriate to line 15. The variable "NUMBER" is equivalent to the variable "Number" which appears in line 13. As previously stated, no distinction is made between upper case and lower case letters.

Line 18 is legal. Again, program readability is enhanced by following the reserved word "end" with the procedure name. There would be a conflict here with the use of the reserved word "new".

Finally, the style (with spaces and blank lines) of the original code is more readable. In general, for good lexical style fit a single statement on one line if possible, and structure compound statements as presented within this workbook, using the indentation recommended (3 spaces).

## EXERCISE 3.2

### OBJECTIVE

To demonstrate the rules for an Ada identifier.

### TUTORIAL

A legal Ada identifier is a name, i.e. a sequence of characters, chosen for types, variables, constants, program units, etc. In Section 2 we saw the following procedure names:

```
Draw_Line  
Draw_Circle  
Display_Square  
Display_Circle
```

and the following function names:

```
Area_Of_Circle  
Area_Of_Square
```

and the following variable names:

```
Top_Left  
Top_Right  
Circle_Area  
Area
```

The language specifies rules for creating legal names. Identifiers not conforming to the language rules are detected at compile time and an appropriate diagnostic is issued. In addition a programmer should create an identifier so that the name chosen accurately describes what is actually represented.

The rules for creating a legal Ada identifier are as follows:

#### An identifier

- starts with a letter
- may be followed by any sequence of letters, digits, and underline characters
- may not contain consecutive underline characters
- has no predefined maximum length yet cannot exceed the length of a single line

The letters used may be either upper case or lower case letters of the alphabet. The upper case and lower case versions of a letter are equivalent in an identifier. The allowable digits consists of the digits 0 through 9. Neither embedded spaces nor special characters (e.g. a question mark) are allowed within an identifier.

The following are legal identifiers:

Simple Graphics	Count
Draw_Line	Head_Count
Area_Of_Square	Page_Count
Circle_Area	Line_Count
Sensor_Switch_1	Beam_Width
Azimuth	beam_width

Notice how the underline improves readability. Sensor\_Switch\_1 is far easier to read than Sensorswitch1. Underlines may be used anywhere within an identifier, except next to another underline or at the beginning or end of an identifier. Remember that no spaces are allowed. Use of an underline to improve readability creates a distinct identifier. Sensor\_Switch\_1 is not the same identifier as Sensorswitch1. Finally, whereas the underline is significant and therefore creates a distinct identifier, the case of a letter is not significant. Beam\_Width and beam\_width are equivalent.

The following are illegal identifiers:

1st Count	-- must start with a letter
head count	-- space illegal
R&D	-- special characters illegal
Not Allowed	-- two adjacent underlines illegal
<u>Also</u> not allowed	-- must start with letter
My_Name_is_	-- trailing underline illegal

### PROBLEM

A point in a plane may be expressed in Cartesian Coordinates or Polar Coordinates. See Figures 3-1 and 3-2 respectively.

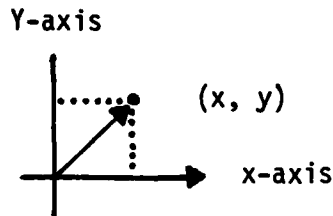


Figure 3-1  
Cartesian Coordinates

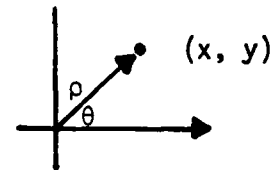


Figure 3-2  
Polar Coordinates

Cartesian Coordinates are an ordered pair of numbers, e.g. (2,3), where the first value represents the x coordinate and the second value represents the y coordinate of a point. The numbers represent the number of units traversed on the x and y axes respectively. Polar Coordinates on the other hand, represent the point in terms of the angle,  $\theta$ , made with the positive x-axis and the distance from the origin,  $\rho$ . The Greek symbols  $\theta$  and  $\rho$  stand for the Greek letters Theta and Rho (pronounced as "row") respectively.

For this exercise choose identifiers to represent the quantities x, y,  $\theta$ , and  $\rho$ .

This page intentionally left blank

## SOLUTION TO EXERCISE 3.2

### SOLUTION

For the Cartesian Coordinate system:

X\_Coordinate  
Y\_Coordinate

For the Polar Coordinate system:

Angle\_From\_X\_Axis  
Length\_From\_Origin

### RATIONALE FOR THE SOLUTION

Use of the identifiers shown above leaves no room for misinterpretation as to what the identifier is to represent.

### ALTERNATIVE SOLUTION

For the Cartesian Coordinate system:

X  
Y

For the Polar Coordinate system:

Rho  
Theta  
R  
T

### DISCUSSION

The alternative solution is less desirable because the identifiers X, Y, R and T could represent a multitude of items. (For instance T could represent either time or target.) Anyone maintaining code containing these identifiers will have to make a conscious effort to remember what these identifiers represent. Additionally an embedded system deals with many angles. Which angle does the identifier "Rho" represent? The angle component of the polar coordinate representation of a potential target or the angle of trajectory of a missile being fired at the target? The identifier "Angle\_From\_X\_Axis" is more appropriate for representing the former while the identifier "Trajectory\_Angle" could be used for the latter.



Finally as shown in Exercise 2.2, the identifiers

Top\_Left  
Top\_Right

are extremely descriptive of the items they are intended to represent.

The point is to choose an identifier appropriate to the item it is intended to represent. Furthermore because Ada does not restrict the length of an identifier as does (for example) FORTRAN, the programmer should name things in full and not use cryptic abbreviations. Proper choice of identifiers is an aid to the production of self-documenting code.

## **Section 4**

### **INTRODUCTION TO DATA**

**4.1 Predefined Integer**

**4.2 Integer Literals**

**4.3 Predefined Float**

**4.4 Real Literals**

**4.5 Type Conversion**

This page intentionally left blank

## EXERCISE 4.1

### OBJECTIVE

To demonstrate Ada's predefined Integer type.

### TUTORIAL

A type in Ada is characterized by a set of values and a set of operations on those values. A type defines the set of possible values that objects of the type (variables and constants) may take on and the set of allowable operations that may be applied to objects of the type. Objects of one type may not be mixed in operations with objects of another type.

The Ada language predefines the type Integer for us. Any object of type Integer has possible integer values within an implementation defined range. Obviously the value of the least allowable integer (most negative) and the largest allowable integer (most positive) depend on the underlying hardware. The values will differ for 16 bit and 32 bit machines, for example.

One format for creating an Integer variable is as follows:

Variable\_Name : Integer;

The name of the variable we wish to create, Variable\_Name, is followed by a colon, ":", which is directly followed by the word Integer to indicate that Variable\_Name is of type Integer. The declaration is terminated by a semi-colon, ";". This declaration creates the object Variable\_Name. By declaring Variable\_Name to be of type Integer, the set of values that Variable\_Name can attain is defined as all whole numbers within the implemented range.

For example, declare three objects X, Y, and Z of type Integer as follows:

X,Y,Z : Integer;

Assuming values are assigned to X and Y, we may add them and assign the result to Z as follows:

Z := X + Y;

Notice that the symbol "!=" represents assignment in Ada.

When declaring an object, an initial value may be assigned to the object. The format for declaring an integer variable with an initial value is as follows:

Variable\_Name : Integer := Initial\_Value;

For example, the statement

X : Integer := 2;

creates the object called X and initially assigns the value 2 to it. Exercise 4.2 addresses the rules for writing integer literals.

As previously stated, a type defines allowable values and allowable operations on those values. The set of operations that can be applied to Variable\_Name are defined below.

#### Binary Numeric Operators

The binary numeric operators allowable on objects and values of type Integer are as follows:

<u>Symbol</u>	<u>Operation</u>
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation
mod	Modulo arithmetic
rem	Remainder on division

#### Binary Relational Operators

The binary relational operators allowable on objects of type Integer are as follows:

<u>Symbol</u>	<u>Operation</u>
=	Equality
/=	Inequality
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

#### Membership Operators

The membership operators allowable on objects of type Integer are as follows:

<u>Symbol</u>	<u>Operation</u>
in	member of
not in	not member of

#### Unary Operators

The unary operators allowable on objects of type Integer are as follows:

<u>Symbol</u>	<u>Operation</u>
+	Plus sign
-	Minus sign
Abs	Absolute Value

Table 4-1 represents a summary of these operations and additionally indicates the result type.

As in FORTRAN, Pascal, etc. Ada expressions are formed using operators, objects, and function calls. Operators are assigned a precedence. Table 4-2 shows the hierarchy of operators. Operators of higher precedence are associated with their operands first. Parentheses may be used to override normal precedence so that operators may be grouped together in any desired way. For expressions containing operators of equal precedence, association of operators to operands proceeds from left to right.

There are cases where it is desirable to limit the range of acceptable values for a specific integer variable. For example, the declaration

```
Cruising_Altitude_In_Feet : Integer;
```

TABLE 4-1  
INTEGER OPERATORS

OPERATORS	RESULT TYPE
<u>Numeric</u>	
+ - * / ** mod rem	Integer
<u>Relational</u>	
= /= < <= > >=	Boolean
<u>Membership</u>	
in not in	Boolean
<u>Unary</u>	
+ - abs	Integer

TABLE 4-2  
HIERARCHY OF OPERATORS

Highest Precedence



Lowest Precedence

parentheses	( )			
highest precedence	**	abs	not	
multiplying	*	/	mod	rem
unary	+	-		
binary	+	-	&	
relational/membership	=	/=	<	<= >
	>=	in	not in	
logical	and	or	xor	and then
	or else			



allows Cruising\_Altitude\_In\_Feet to take on any integer value in the implementation defined range. Some of those values are meaningless, e.g., 10 and 500,000. Planes do not cruise at either 10 feet or 500,000 feet. Ada allows the programmer to restrict the range of values that an integer variable may acquire by adding a range constraint to the declaration.

A range constraint has the following form:

range L .. R

where

- "range" is a reserved word
- L represents the lower bound
- R represents the upper bound

If an operation attempts to assign a value which is  $< L$  or  $> R$  to an object with this range constraint, then suspension of normal program execution occurs. For now it suffices to say that values not satisfying a specific range constraint cannot be applied to objects that have been declared with that range constraint.

This leads to a third format for Integer object declarations which is as follows:

Variable\_Name : Integer range L .. R;

or

Variable\_Name : Integer range L .. R := Initial\_Value;

As an example, the following declaration

Cruising\_Altitude\_In\_Feet : Integer range 15000 .. 40000;

creates the Integer objects Cruising\_Altitude\_In\_Feet and restricts the possible set of values to those whole numbers between 15000 and 40000 inclusive. If at run time, Cruising\_Altitude\_In\_Feet is assigned a value outside the range constraint, e.g. 50,000, the program containing this statement would abort.

# PROBLEM

Text formatters have pagination capabilities. In order to print a new page number, the number of lines printed per page must be counted. Declare an object of type Integer to maintain the line count and initialize it to zero (0). Assume the page is defined to accommodate 70 lines.

This page intentionally left blank

## SOLUTION TO EXERCISE 4.1

### SOLUTION

The object declaration with the required initialization is:

```
Line_Count : Integer range 0 .. 70 := 0;
```

### RATIONALE FOR THE SOLUTION

The object desired, Line Count, is followed by the required colon, which is then directly followed by the name of the type, Integer. The optional range constraint

```
range 0 .. 70
```

follows the type name and consists of the keyword "range" followed by the lower bound and upper bound separated by two dots. Finally, the declaration is terminated by the required semicolon.

### ALTERNATIVE SOLUTION

The following might have been specified for the object declaration:

```
Line_Count : Integer := 0;
```

### DISCUSSION

Both object declarations specify that we have created a variable called Line\_Count which can take on only integer values and can be operated on by the set of operations defined by the language. Specifically, after printing a line, Line\_Count may be incremented as in:

```
Line_Count := Line_Count + 1;
```

Line\_Count may be compared to 70 as in the following expression which returns a Boolean result:

```
Line_Count = 70
```

The declaration as presented in the solution is preferable because the specified range constraint restricts the values that Line\_Count can attain to the whole numbers 0 through 70 inclusive. The range constraint makes it impossible for the identifier Line\_Count to take on any value outside the range specified, e.g. 72. Range constraints don't forbid the computation of Line\_Count + 1 when Line\_Count = 70, but do forbid the assignment of the resulting value to Line\_Count. Any attempt to assign a value to Line\_Count at runtime which is outside the range will cause the program to abort.

The lack of the range constraint in the alternative solution allows Line\_Count to attain any legal integer value. Specifically Line\_Count can take on a value such as 564 which is meaningless in terms of our definition of the maximum number of lines allowed per page as 70.

## EXERCISE 4.2

### OBJECTIVE

To demonstrate the rules for writing integer literals.

### TUTORIAL

The notation used to represent a specific value is called a literal. A specific example of an Integer literal is required in the previous exercise as an initial value to `Line_Count`. 0 is a literal, specifically a numeric literal.

Numeric literals in Ada are of two classes, integer literals and real literals. Integer literals, like 2, represent whole numbers. Real literals, like 3.3, represent approximations and are not usually whole numbers. This exercise addresses integer literals only. Exercise 4.4 addresses real literals.

Integer literals are normally assumed to be of base 10. The language does allow for the specification of integer literals in a base other than 10.

The rules for writing an integer literal are as follows:

An integer literal of base 10

- is written without a decimal point
- uses the digits 0 through 9
- may be written in exponential notation
- may contain isolated embedded underlines between consecutive digits

In exponential notation the exponent indicates the power of ten that the value of the literal without the exponent (i.e., the mantissa) is to be multiplied by to obtain the value of the literal. In Ada, one writes the integer literal directly followed by the letter "E" (or "e") directly followed by the positive power of 10. An optional + sign may be inserted between the E and the power of 10. A - sign may not.

In order to represent a negative integer, precede the integer literal by a minus sign. A negative integer is thus not a literal but an expression. For a positive integer literal the plus sign is optional.

The following are legal integer literals:

1	-- represents one (1)
100	-- represents one hundred (100)
1E2	-- represents one hundred (100)
	-- capital E allowed
1e2	-- represents one hundred (100)
	-- lower case e allowed
1e+2	-- represents one hundred (100)
10E1	-- represents one hundred (100)
123456	-- hard to read
123_456	-- easier to read, represents same
	-- value as 123456
12E6_6	-- underline legal within exponent

Use of the underline in creating numeric literals is not significant. Whereas Sensor\_Switch 1 and Sensorswitch1 are two distinct identifiers, 123456 and 123\_456 represent the same integer literal.

The following are illegal integer literals:

1.2	-- decimal point illegal
1e-6	-- negative exponent illegal
123_456	-- adjacent underlines illegal
1e 6	-- space illegal
123 456	-- space illegal
123 456	-- trailing underline illegal
123,456	-- comma illegal
-1	-- negative integer is an
	-- expression, not a literal

PROBLEM

Find and correct the error(s) in the following package specification.

```
package Some_Conversion_Constants is
    Meter_To_Decimeter : Integer := 100e-1;
    Meter_To_Centimeter : Integer := 100.;
    Meter_To_Kilometer : Integer := 1,000;
end Some_Conversion_Constants;
```



This page intentionally left blank

## SOLUTION TO EXERCISE 4.2

### SOLUTION

```
package Some_Conversion_Constants is

    Meter_To_Decimeter : Integer := 100e-1;  -- ERROR
    Meter_To_Centimeter : Integer := 100.;    -- ERROR
    Meter_To_Kilometer  : Integer := 1,000;   -- ERROR

end Some_Conversion_Constants;
```

The corrected code is

```
package Some_Conversion_Constants is

    Meter_To_Decimeter : Integer := 10;
    Meter_To_Centimeter : Integer := 100;
    Meter_To_Kilometer  : Integer := 1000;

end Some_Conversion_Constants;
```

### RATIONALE FOR THE SOLUTION

The literals are illegal for the following specified reasons:

100e-1	-- negative exponent not allowed
100.	-- decimal not allowed
1,000	-- commas not allowed

The corrected code uses the fact that integer literals are written without decimal points and may only contain isolated embedded underlines between consecutive digits.

### ALTERNATIVE SOLUTION

An alternative way of correcting the code is:

```
package Some_Conversion_Constants is
```

```
    Meter_To_Decimeter : Integer := 1E1;
```

```
    Meter_To_Centimeter : Integer := 1E2;
```

```
    Meter_To_Kilometer : Integer := 1E3;
```

```
end Some_Conversion_Constants;
```

#### DISCUSSION

There is no real preferred solution here. The issue is readability and it really is a value judgment as to which style a programmer prefers.

## EXERCISE 4.3

### OBJECTIVE

To demonstrate Ada's predefined Floating Point type.

### TUTORIAL

In addition to predefining the integer type Integer, the Ada language predefines a floating point data type called Float. This is the same type Float that we have seen used in previous exercises. The range of acceptable floating point values is implementation dependent.

Real data types are approximations only. It is not possible to provide exact machine representations for all real values, e.g., those values that have repeating binary fractional parts such as .1. Therefore, all real values of type Float are represented in terms of an error bound which is explicitly stated for each implementation. In other words, the accuracy of the predefined type Float is implementation dependent.

The format for creating a floating point variable is as follows:

Variable\_Name : Float;

Where Variable\_Name is an identifier written according to the rules specified in Exercise 3.2.

When declaring a variable, an initial value may be assigned. The format for declaring a floating point variable with an initial value is as follows:

Variable\_Name : Float := Initial\_Value;

For example, the statement

X : Float := 3.0;

creates the object called X and initially assigns the value 3.0 to it. Exercise 4.4 addresses the rules for writing real literals such as 3.0.

Table 4-3 depicts the operations that can be performed on objects of type Float. Be advised that the right operand for exponentiation must be of

TABLE 4-3

## LEGAL FLOATING POINT OPERATIONS

OPERATORS	RESULT TYPE
<u>Numeric</u>	
+ - * / **	Float
<u>Relational</u>	
= /= < <= > >=	Boolean
<u>Unary</u>	
+ - abs	Float

an integer type. The operations are subject to the same hierarchy discussed in Exercise 4.1 and illustrated in Table 4-2.

Examine the following function which converts degrees to radians.

```
function Convert_Degrees_To_Radians (Degrees : Integer)
  return Float is

  Pi : constant Float := 3.1415923;

begin -- Convert_Degrees_To_Radians

  return Degrees * Pi/180.0; -- ERROR

end Convert_Degrees_To_Radians;
```

Let's discuss the error in the return statement. The function specification indicates that the type of the value to be returned is Float. The expression in the return statement consists of the Floating point object Pi, the real literal 180.0, and the Integer object Degrees. This expression is illegal. Objects of different types may not be combined in expressions. Obviously one would like to be able to convert degrees to radians (and vice versa). If the original function specification had specified the parameter to be of type Float as follows:

```
function Convert_Degrees_To_Radians (Degrees : Float)
  return Float is

  Pi : constant Float := 3.1415923;

begin -- Convert_Degrees_To_Radians

  return Degrees * Pi/180.0; -- legal

end Convert_Degrees_To_Radians;
```

then the expression in the return statement becomes legal. Exercise 4.5 illustrates simple type conversion which allows for another solution.

This page intentionally left blank

### PROBLEM

Find and correct the errors in the following procedure. (Hint: See the specification for package Simple\_Graphics.)

```
with Simple_Graphics; use Simple_Graphics;
procedure Draw_Specific_Square is

    -- This procedure will draw a
    -- square with top left
    -- corner at (0, 0) and
    -- side length 3.

begin -- Draw_Specific_Square

    Draw_Line ((0, 0), (3, 0));
    Draw_Line ((3, 0), (3, -3));
    Draw_Line ((3, -3), (0, -3));
    Draw_Line ((0, -3), (0, 0));

end Draw_Specific_Square;
```



This page intentionally left blank

## SOLUTION TO EXERCISE 4.3

### SOLUTION

```
with Simple_Graphics; use Simple_Graphics;
procedure Draw_Specific_Square is

    -- This procedure will draw a
    -- square with top left
    -- corner at (0,0) and
    -- side length 3.

begin -- Draw_Specific_Square

    Draw_Line ((0, 0), (3, 0));      -- ERROR
    Draw_Line ((3, 0), (3, -3));     -- ERROR
    Draw_Line ((3, -3), (0, -3));    -- ERROR
    Draw_Line ((0, -3), (0, 0));     -- ERROR

end Draw_Specific_Square;
```

The correct solution is:

```

with Simple_Graphics; use Simple_Graphics;
procedure Draw_Specific_Square is

    -- This procedure will draw a
    -- square with top left
    -- corner at (0.0, 0.0) and
    -- side length 3.

begin
    -- Draw_Specific_Square

    Draw_Line ((0.0, 0.0), (3.0, 0.0));
    Draw_Line ((3.0, 0.0), (3.0, -3.0)); -- all values
    Draw_Line ((3.0, -3.0), (0.0, -3.0)); -- have decimals
    Draw_Line ((0.0, -3.0), (0.0, 0.0));

end Draw_Specific_Square;

```

#### RATIONALE FOR THE SOLUTION

The procedure specification for Draw\_Line in the package specification for Simple\_Graphics requires two parameters each of type Point. Point is declared to be an array of 2 components each of type Float. In the statement of the problem, the four calls to Draw\_Line in procedure Draw\_Specific\_Square have integer components for Point. These four calls are illegal subprogram calls as the types of the formal and actual parameters do not match.

The corrected code uses decimal points.

#### ALTERNATIVE SOLUTION

There is no alternative solution to this exercise.

#### DISCUSSION

As previously stated, Ada is a strongly typed language. All objects must be declared, all expressions must be of one type, all formal and actual parameters must agree in number and in type. Point is declared to be an array with 2 components of type Float. Any parameters passed to Draw\_Line must be of type Float. The literals in this code require a decimal point.

## EXERCISE 4.4

### OBJECTIVE

To demonstrate the rules for writing real literals.

### TUTORIAL

As previously stated, numeric literals in Ada are of two classes, integer literals and real literals. Integer literals represent whole numbers, like 3, and were addressed in Exercise 4.2. Real literals represent approximations, like 3.3. Real literals always include a decimal point.

As with integer literals, real literals are normally assumed to be of base 10. It is possible to represent real values in bases other than 10.

The rules for writing real literals are as follows:

A real literal

- is written with a decimal point
- uses the digits 0 through 9
- must have a digit on either side of the decimal point
- may contain a single embedded underline between consecutive digits
- may be written in exponential notation with either positive or negative exponents

Exponential notation for real literals is essentially identical to exponential notation for integer literals with the exception that real literals may have negative exponents.

In exponential notation the exponent indicates the power of ten that the value of the literal without the exponent (i.e. the mantissa) is to be multiplied by to obtain the value of the literal. In Ada, one writes the real literal directly followed by the letter "E" (or "e") directly followed by the power of 10. An optional + or - sign may be inserted between the E and the power of 10.

Neither embedded spaces nor commas are allowed in real literals.

The following are legal real literals:

300.0	-- represents 300
3.OE2	-- represents 300
3.Oe2	-- represents 300
3.OE-2	-- represents .03
1000000.0	-- represents one million
1_000_000.0	-- represents one million
1.OE6	-- represents one million
1.Oe6	-- represents one million

As in Integer literals, use of underlines in writing real literals does not create a distinct value. 1000000.0 and 1\_000\_000.0 are alternative notations each representing the same value.

The following are illegal real literals:

1,000,000.0	-- commas illegal
1 000 000.0	-- spaces illegal
1_000_000.0	-- adjacent underlines illegal
2_500.0_	-- trailing underline illegal
300	-- decimal point missing
.333	-- digit missing before decimal point
33.	-- digit missing after decimal point
1_.0	-- underline can occur only between consecutive digits.
1.OE2.3	-- exponent must be integer

PROBLEM

Write the number 1,000 as a decimal real literal in as many ways as possible without an underline.

This page intentionally left blank

## SOLUTION TO EXERCISE 4.4

### SOLUTION

1000.0	
1.0E3	1.0E+3
1.0e3	1.0e+3
10.0E2	10.0E+2
10.0e2	10.0e+2
100.0E1	100.0E+1
100.0e1	100.0e+1
1000.0E0	1000.0E+0
1000.0e0	1000.0e+0

### RATIONALE FOR THE SOLUTION

The table depicted above indicates a progression of representations for 1,000 using positive exponents, with and without the optional plus signs. Additionally, the exponent is written in both uppercase "E" and lowercase "e."

The solution does not include those values that can be formed utilizing optional underlines such as 1\_000.0E0 or 10\_0.0E1, or those values that can be obtained using negative exponents such as 10000.0E-1. Additionally, the solution does not contain representations such as 0.1E+7 nor representations such as 10#1000.0#. To illustrate the total number of combinations possible is far too lengthy, i.e. infinite!

### ALTERNATIVE SOLUTION

There is no alternative solution for this exercise.

### DISCUSSION

The choice of expression for a specific value is necessarily a function of its readability and space limitations on a line. For instance, one million expressed as 10.0e5 is not as immediately recognizable as 1.0e6. Furthermore, 1.0e6 is far more compact than 1\_000\_000.0. An attempt should be made to represent the value desired in as straightforward a manner as possible and in such a fashion that the eye can grasp the intended value in one glance. It takes one eye movement to recognize 1.0e6, while it takes three eye movements to recognize 1\_000\_000.0 because the zeros must be counted.



Additionally, use underlines intelligently. Place them where you would normally use a comma. For example, though they represent the same number, 1\_000.0 is a far more appropriate representation for 1,000.0 than 10\_00.0.

## EXERCISE 4.5

### OBJECTIVE

To introduce type conversion for numeric types.

### TUTORIAL

Ada is a strongly typed language. As we saw in Exercise 4.3, operations on objects of different types are prohibited. Floating point and Integer objects cannot be mixed within one expression.

Actually Ada does allow mixing of objects of different types but in a structured and controlled fashion. Objects of some types may be converted to objects of other types but that conversion must be explicit. By requiring an explicit conversion, a maintainer of an Ada program is made aware that this value is being used in an unusual fashion.

Explicit type conversions are allowed between numeric types. Conversion from Float to Integer is obtained by rounding to the nearest integer.

The format for a type conversion is as follows:

```
Integer (F)
Float (I)
```

where:

```
F is a floating point expression
I is an integer expression
```

Recall procedure `Square_Root` introduced in Section 1. It contains the statement

```
Put (Sqrt (X));
```

where `Sqrt` is a function located in package `Math_Lib` which returns the square root of the parameter passed to it. Examining the code further, we see that this parameter is of type `Float`.

Now consider the following incorrect Ada program, which performs various calculations on some integer objects and assigns the result to the integer object `Intermediate_Result`. Finally the square root of `Intermediate_Result` is assigned to the floating point object `Final_Result`.

```

with Math_Lib; use Math_Lib;
with Simple_IO; use Simple_IO;
procedure Main_Proc is

    Intermediate_Result : Integer;
    Final_Result        : Float;

begin -- Main_Proc

    -- Some calculations to obtain Intermediate_Result

    -- Square root of Intermediate_Result assigned to
    -- Final_Result.

    Final_Result := Sqrt (Intermediate_Result); -- ERROR
    Put (Final_Result);

end Main_Proc;

```

The statement

```
Final_Result := Sqrt (Intermediate_Result);
```

is not legal. We are trying to supply an object of type Integer where an object of type Float is required. Type conversion is needed here. The correct statement is

```
Final_Result := Sqrt (Float (Intermediate_Result));
```

The expression `Float (Intermediate_Result)` converts the value of the object `Intermediate_Result` to a floating point value before it is passed to the square root function.

### PROBLEM

Very rarely, if at all, does a system operator see a target identified or target tracking information displayed on a screen with necessary angles identified in radians. Angles are usually reported to a human operator in degrees. Yet frequently, functions such as sine and cosine require their parameters in terms of radians.

Assume package Math\_Lib exists with the following minimal specifications.

```
package Math_Lib is

  function Convert_Degrees_To_Radians (Angle_In_Degrees : Float)
    return Float;
  function Sine (Angle_In_Radians : Float) return Float;

  -- Possibly other subprograms declared here.

end Math_Lib;
```

Complete the code below which computes the sine of a trajectory angle.

```
with Math_Lib; use Math_Lib;
with Simple_IO; use Simple_IO;
procedure Sine_Of_Traj_Angle is

  -- This procedure obtains the value of Traj_Angle_Degrees,
  -- computes the sine, and outputs the value.

  Traj_Angle_Degrees : Integer;
  Traj_Angle_Radians : Float;
  Sine_Traj_Angle    : Float;

begin -- Sine_Of_Traj_Angle

  -- Get angle in degrees

  Get (Traj_Angle_Degrees);

  -- Convert degrees to radians

  -- Compute Sine

  -- Output value

  Put (Sine_Traj_Angle);

end Sine_Of_Traj_Angle;
```

This page intentionally left blank

## SOLUTION TO EXERCISE 4.5

### SOLUTION

```
with Math_Lib; use Math_Lib;
with Simple_IO; use Simple_IO;
procedure Sine_Of_Traj_Angle is

    -- This procedure obtains the value of Traj_Angle_Degrees,
    -- computes the sine, and outputs the value.

    Traj_Angle_Degrees : Integer;
    Traj_Angle_Radians : Float;
    Sine_Traj_Angle    : Float;

begin -- Sine_Of_Traj_Angle

    -- Get angle in degrees
    Get (Traj_Angle_Degrees);

    -- Convert degrees to radians
    Traj_Angle_Radians :=
        Convert_Degrees_To_Radians (Float (Traj_Angle_Degrees));

    -- Compute Sine
    Sine_Traj_Angle := Sine (Traj_Angle_Radians);

    -- Output value
    Put (Sine_Traj_Angle);

end Sine_Of_Traj_Angle;
```

#### RATIONALE FOR THE SOLUTION

The function `Convert_Degrees_To_Radians` in `Math_Lib` expects a parameter of type `Float` which represents the value in degrees of the angle to be converted. `Traj_Angle_Degrees` is of type `Integer` and must be converted to `Float` before being passed to `Convert_Degrees_To_Radians`, hence the requirement for the type conversion

`Float (Traj_Angle_Degrees)`

within the statement

```
Traj_Angle_Radians :=  
  Convert_Degrees_To_Radians (Float (Traj_Angle_Degrees));
```

Now `Sine_Traj_Angle` may be computed as follows:

```
Sine_Traj_Angle := Sine (Traj_Angle_Radians);
```

`Traj_Angle_Radians` is of type `Float` and represents an angle expressed in radians. That is precisely what the `Sine` function expects as a parameter.

### ALTERNATE SOLUTION

```
with Math_Lib; use Math_Lib;
with Simple_IO; use Simple_IO;
procedure Sine_Of_Traj_Angle is

    -- This procedure obtains the value of Traj_Angle_Degrees
    -- from the screen, computes the sine, and outputs the value.

    Traj_Angle_Degrees : Integer;
    Sine_Traj_Angle     : Float;

begin -- Sine_Of_Traj_Angle

    -- Get angle in degrees

    Get (Traj_Angle_Degrees);

    -- Compute Sine

    Sine_Traj_Angle :=
        Sine (Convert_Degrees_To_Radians (Float (Traj_Angle_Degrees)));

    -- Output value

    Put (Sine_Traj_Angle);

end Sine_Of_Traj_Angle;
```

### DISCUSSION

The alternative solution leads to a discussion of the tradeoffs between introducing local data and readability. Although the alternative solution requires one less local data item and one less executable statement, the remaining executable statement

```
Sine_Traj_Angle :=
    Sine (Convert_Degrees_To_Radians (Float (Traj_Angle_Degrees)));
```

is cumbersome and time consuming to decipher. In this particular instance introduction of local data improves the readability of the code, which leads to more maintainable code.



This page intentionally left blank

## **Section 5**

### **ENUMERATION: TYPES AND CONTROL STRUCTURES**

**5.1 Enumeration Types/Objects**

**5.2 Case Statement**

**5.3 Boolean Types/Objects**

**5.4 If Statement**

This page intentionally left blank

## EXERCISE 5.1

### OBJECTIVE

To illustrate enumeration types.

### TUTORIAL

The language supplies the predefined types Integer and Float introduced in Section 4. These are numeric types and, therefore, objects of these types are restricted to contain numeric values. What does one do if it is desirable to talk about the ranks in the Army, the status of a communication link, or more simply the days of the week? These elements are talked about not in terms of numerics but in terms of "words." For example, the ranks of the Army are Private, Sergeant, Lieutenant, etc.; link status is up, down, or busy; the days of the week are Sunday, Monday, Tuesday, etc. In most languages the traditional approach is to implement the ranks of the Army and the days of the week in terms of numeric codes (e.g., 1 for Private, 2 for Sergeant, etc.) and then consciously remember which code means Lieutenant, or which means Tuesday, and so on.

Ada offers another class of types, called enumeration types, which allows the programmer to explicitly list (enumerate) the legal set of values for a type in an ordered list. An enumeration type is not a predefined type but rather a user-defined type.

The format for an enumeration type declaration is as follows:

```
type Type_Name is (Value_1, Value_2, ... Value_n);  
where  
• Type_Name is a user-supplied identifier for the type being  
  introduced  
• Value_i is either an identifier or a character literal*  
• the list of allowed values for the type is enclosed  
  in parentheses and has an implied order
```

Enumeration type declarations are terminated by a semicolon. The specified values are called enumeration literals.

\*A character literal is written by enclosing one member of the ASCII character set within single apostrophes, e.g., 'R'.

The following are legal enumeration type declarations:

```
type Army_Rank is (Private, Sergeant, Lieutenant, Captain);
type Link_Status is (Up, Down, Busy);
type Days_Of_the_Week is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
type Hex_Digit is ('0', '1', '2', '3', '4', '5', '6', '7', '8',
                   '9', 'A', 'B', 'C', 'D', 'E', 'F');
```

An enumeration type declaration 1.) names the type, 2.) defines the set of allowable values to be associated with objects of the type, and 3.) implicitly associates with these values a set of operations applicable to objects of the type.

Enumeration variables are declared as follows:

Variable\_Name : Enumeration\_Type\_Name;

The user-supplied identifier to be associated with the object, Variable\_Name, is followed by a colon, which in turn is directly followed by the type name. The declaration is terminated by a semicolon. Here are some sample object declarations:

```
Today          : Days_Of_the_Week;
Status_Link_1  : Link_Status;
Status_Link_2  : Link_Status;
```

Operations on objects of an enumeration type are listed in Table 5-1. The relational operators are a consequence of the fact that the values specified in the type declaration (i.e., the enumeration literals) for a specific enumeration type are ordered. For example, Mon < Tue is True, while Mon > Tue is False.

Consider the following object declaration:

```
Today : Days_Of_The_Week;
```

Today could have been initialized as in

```
Today : Days_Of_The_Week := Mon;
```

TABLE 5-1  
ENUMERATION OPERATORS

OPERATORS	RESULT TYPE
<u>Relational</u>	
=    / =    <    < =    >    > =	Boolean
<u>Membership</u>	
in    not in	Boolean

and it may be assigned to as in

```
Today := Sun;
```

Any attempt to initialize or assign to Today a value other than those listed in the type declaration for Days\_Of\_The\_Week is illegal. For example, the assignment

```
Today := 1;
```

will be rejected by the compiler.

Finally, Today may be compared with the enumeration literals of Days\_Of\_The\_Week or other objects of type Days\_Of\_The\_Week, yielding a Boolean value, as in

```
(Today < Tue)
```

The value of the expression Today < Tue is True only if Today has a value of Sun or Mon, and is False otherwise. Also,

```
Holiday : Days_Of_The_Week := Mon;
```

```
(Today < Holiday)
```

is True because Today has the value of Sun and Holiday has the value of Mon.

It is important to realize the difference between types and objects. The type declaration

```
type Link_Status is (Up, Down, Busy);
```

is an enumeration type declaration. It defines a type named Link\_Status and states that the only legal values for objects of type Link\_Status are Up, Down, and Busy and the only allowable operations are those listed in Table 5-1. Objects of type Link\_Status must be created just as objects of type Integer and Float are created. The type declaration defines the set of values that objects of type Link\_Status may take on, but does not

create the object. To create an object, an object declaration is required as in

```
Local_Link_Status : Link_Status;
```

or

```
Remote_Link_Status : Link_Status;
```

The object `Local_Link_Status` may be assigned one of the values `Up`, `Down`, or `Busy`; the relational operators or membership operators may be applied, or it may be passed as a parameter. These operations cannot be performed on `Link_Status`.

As a final note, identical enumeration literals may be used in distinct enumeration type declarations. For example the two enumeration type declarations

```
type Storage_Unit is (Bit, Byte, Word);  
type Message_Type is (Character, Bit);
```

both contain a value `Bit`. The enumeration literal `Bit` is said to be overloaded. The following object declaration

```
JANAP_128_Message : Message_Type := Bit;
```

takes the value `Bit` from the declaration of type `Message_Type`. There is no ambiguity as to which `Bit` is required.



This page intentionally left blank

### PROBLEM

A requirement of a message switch is that it should handle various transmissions on different channels. Each transmission type is described in terms of channel modes. Consider a switch that must support the following channel modes:

<u>Channel Mode</u>	<u>Transmission Type</u>
1	Synchronous, Blocked
2	Asynchronous, Unidirectional
3	Synchronous, Continuous
4	Asynchronous, Bidirectional
5	Asynchronous, Unidirectional

Declare an enumeration type and object named Channel\_Mode\_Type and Message\_Switch\_Channel\_Mode, respectively, to represent the requirements for this message switch.

This page intentionally left blank

## SOLUTION TO EXERCISE 5.1

### SOLUTION

The required type declaration is

```
type Channel_Mode_Type is (Channel_Mode_1, Channel_Mode_2,  
                           Channel_Mode_3, Channel_Mode_4,  
                           Channel_Mode_5);
```

The required object declaration is

```
Message_Switch_Channel_Mode : Channel_Mode_Type;
```

### RATIONALE FOR THE SOLUTION

The type declaration has the required word "type," followed by the name of the type, Channel\_Mode\_Type, directly followed by the reserved word "is." The enumeration literals are enclosed in parentheses and separated by commas. Each value is a legal Ada identifier. Finally, the declaration is terminated by the required semicolon.

The object declaration has the required format. The name of the object to be created, Message\_Switch\_Channel\_Mode, is followed by the required colon which is then directly followed by the type, Channel\_Mode\_Type. The required semicolon terminates the declaration.

### ALTERNATIVE SOLUTION

```
type Channel_Mode_Type is (One, Two, Three, Four, Five);
```

### DISCUSSION

The alternative type declaration is syntactically correct as each of the values is an identifier. However, the enumeration literals do not convey as much information as in the declaration presented in the solution. The values as presented in the solution are a better abstraction of the real world. Correct usage of enumeration types and enumeration values is imperative for the production of readable and maintainable code.

This page intentionally left blank

## EXERCISE 5.2

### OBJECTIVE

To illustrate the case statement.

### TUTORIAL

For conditional control, Ada provides the if statement and the case statement. The if statement is addressed in Exercise 5.4. The case statement is the topic of this exercise.

The case statement selects statements for execution from alternative sequences of statements based on the value of a discrete expression. A discrete expression evaluates to a discrete value, i.e., a value that is either an integer or an enumeration literal.

The format for a case statement is as follows:

```
case Discrete_Expression is
    when Choice_1 =>
        statement(s);
    when Choice_2 =>
        statement(s);
    .
    .
    when Choice_n =>
        statement(s);
    when others =>
        statement(s); -- optional
end case;
```

where

- Discrete\_Expression is any legal discrete expression
- Choice\_1 through Choice\_n represent mutually exclusive and exhaustive values of the same type as the expression
- The optional choice, "when others," represents the choice for those values of the type not specifically covered in previous choices. "When others," if used, must appear last.
- When Choice\_1 through Choice\_n is not exhaustive, "others" must be present.

Let's revisit the problem of Exercise 5.1. The transmission is synchronous for channels 1 and 3, asynchronous for channels 2, 4, and 5. The following procedure uses a case statement to select the correct processing mode for a channel.

```

procedure Select_Correct_Processor (Channel_Mode : in Channel_Mode_Type) is
    procedure Process_Synch is separate;
    procedure Process_Asynch is separate;

begin -- Select_Correct_Processor

    case Channel_Mode is
        when Channel_Mode_1 =>
            Process_Synch;
        when Channel_Mode_2 =>
            Process_Asynch;
        when Channel_Mode_3 =>
            Process_Synch;
        when Channel_Mode_4 =>
            Process_Asynch;
        when Channel_Mode_5 =>
            Process_Asynch;
    end case;

end Select_Correct_Processor;

```

The Discrete\_Expression, Channel\_Mode, is a discrete expression. Its value is an enumeration value. The choices represented in the case statement are mutually exclusive, i.e., only one choice is possible for each value of the specified type. The choices are exhaustive, i.e., they represent all possible values of type Channel\_Mode\_Type, so an others choice is not necessary.

There are alternative notations for expressing the fact that the same sequence of statements is to be executed for multiple choices. The choices may be separated by a bar as follows:

```

when Choice_1 | Choice_2 => statement(s);

```

Here when Discrete\_Expression evaluates to either Choice\_1 or Choice\_2, the specified statement(s) are executed.

For example, Select\_Correct\_Processor could be written as

```

procedure Select_Correct_Processor (Channel_Mode : in Channel_Mode_Type) is

    procedure Process_Synch is separate;
    procedure Process_Asynch is separate;

begin -- Select_Correct_Processor

    case Channel_Mode is
        when Channel_Mode_1 | Channel_Mode_3 =>
            Process_Synch;
        when Channel_Mode_2 | Channel_Mode_4 |
            Channel_Mode_5 =>
            Process_Asynch;
    end case;

end Select_Correct_Processor;

```

When Channel\_Mode evaluates to either Channel\_Mode\_1 or Channel\_Mode\_3, procedure Process\_Synch is invoked. Procedure Process\_Asynch is invoked when Channel\_Mode evaluates to either Channel\_Mode\_2, Channel\_Mode\_4, or Channel\_Mode\_5.

Or the choices may be expressed as a discrete range as in:

```

when Choice_i .. Choice_j => statement(s);

```

In this latter format, the choices must be consecutive values. Here when Discrete\_Expression evaluates to any value in the range Choice\_i .. Choice\_j, the specified statement(s) are executed. For example, Select\_Correct\_Processor could be coded as

```

procedure Select_Correct_Processor (Channel_Mode : in Channel_Mode_Type) is

    procedure Process_Synch is separate;
    procedure Process_Asynch is separate;

begin -- Select_Correct_Processor

    case Channel_Mode is
        when Channel_Mode_1 | Channel_Mode_3 =>
            Process_Synch;
        when Channel_Mode_2 | Channel_Mode_4 ..
            Channel_Mode_5 =>
            Process_Asynch;
    end case;

end Select_Correct_Processor;

```



Whether the choices Channel\_Mode\_4 and Channel\_Mode\_5 are separated by a bar or expressed as a discrete range is a matter of personal preference. Here, procedure Process\_Synch is invoked when Channel\_Mode evaluates to either Channel\_Mode\_1 or Channel\_Mode\_3, and procedure Process\_Asynch is executed otherwise. A final view of Select\_Correct\_Processor is

```
procedure Select_Correct_Processor (Channel_Mode : in Channel_Mode_Type) is
    procedure Process_Synch is separate;
    procedure Process_Asynch is separate;

begin -- Select_Correct_Processor

    case Channel_Mode is
        when Channel_Mode_1 | Channel_Mode_3 =>
            Process_Synch;
        when others =>
            Process_Asynch;
    end case;

end Select_Correct_Processor;
```

The choice "when others" appears last and includes all possible values for Channel\_Mode not specified explicitly in previous choices.

Each of the above examples has a single procedure call as the sequence of statements to be executed for a specific choice. This need not be the case. Any legitimate sequence of statements may be executed for a specific choice. As an example, suppose Channel\_Mode\_1 only carries messages of extreme urgency. Upon receipt of a message on Channel\_Mode\_1 a message is displayed to the operator indicating arrival of an urgent message. The procedure now takes the form

```

with Text_IO; use Text_IO;
procedure Select_Correct_Processor (Channel_Mode : in Channel_Mode_Type) is

    procedure Process_Synch is separate;
    procedure Process_Asynch is separate;

begin -- Select_Correct_Processor

    case Channel_Mode is
        when Channel_Mode_1 =>
            Put ("Urgent Message");
            Process_Synch;
        when Channel_Mode_3 =>
            Process_Synch;
        when Channel_Mode_2 | Channel_Mode_4 |
            Channel_Mode_5 =>
            Process_Asynch;
    end case;

end Select_Correct_Processor;

```

The following summarizes the rules for the case statement:

- the expression must be discrete
- there must be one, and only one possible alternative for each possible value of the expression
- choices may be separated by a bar (|)
- choices may be specified in a discrete range
- "when others" is optional yet when appearing must be last

This page intentionally left blank

### PROBLEM

One requirement of a message switch is to route assembled messages to their assigned destination. Each message contains a field called a routing indicator (RI) field which contains a code for the required destinations. Usually the first letter of the RI field indicates whether the field is a legitimate field or not. Let us assume that for this switch, the only legal first characters for the RI field are R, U, and Y.

Using a case statement, write a procedure named Check\_RI which

- checks the first character of the RI field and calls procedure Route\_Message if the character is legal, otherwise calls another procedure named Error\_Routine
- sets the value of an object Status of type Status\_Type to its appropriate value. Type Status\_Type is defined as

type Status\_Type is (OK, Error);

The specification for Check\_RI is as follows:

```
procedure Check_RI (RI_Field : in Character;
                    Status    : out Status_Type)
```

Assume that Status\_Type is available to Check\_RI. (Hint: The language predefined for us the types Integer and Float. Additionally type Character is predefined by the language. Its set of legal values are the 128 members of the ASCII character set. Operations allowed on objects of type Character are those listed in Table 5-1. Remember, character literals are written by enclosing one member of the ASCII character set within single apostrophes, e.g., 'R'.)

This page intentionally left blank

## SOLUTION TO EXERCISE 5.2

### SOLUTION

```
procedure Check_RI (RI_Field : in Character;
                    Status    : out Status_Type) is

    procedure Route_Message is separate;
    procedure Error_Routine is separate;

begin -- Check_RI

    case RI_Field is
        when 'R' | 'U' | 'Y' =>
            Status := OK;
            Route_Message;
        when others =>
            Status := Error;
            Error_Routine;
    end case;

end Check_RI;
```

### RATIONALE FOR THE SOLUTION

The procedure has the two required parameters; the "in" parameter RI\_Field of type Character and the "out" parameter Status of type Status\_Type. The two stubs for procedures Route\_Message and Error\_Routine allow us to defer production of their actual code. In the case statement, RI\_Field is a discrete expression since RI\_Field is of type Character which is a predefined enumeration type. If the value of RI\_Field evaluates to either R, U, or Y, then Status is assigned the value OK and Route\_Message is called. For all other values of type Character, Status is assigned the value Error and Error\_Routine is invoked. The alternative "when others" appears last as required.

### ALTERNATIVE SOLUTION

```
procedure Check_RI (RI_Field : in Character;  
                    Status    : out Status_Type) is  
  
    procedure Route_Message is separate;  
    procedure Error_Routine is separate;  
  
begin -- Check_RI  
  
    case RI_Field is  
        when 'R' =>  
            Status := OK;  
            Route_Message;  
        when 'U' =>  
            Status := OK;  
            Route_Message;  
        when 'Y' =>  
            Status := OK;  
            Route_Message;  
        when others =>  
            Status := Error;  
            Error_Routine;  
    end case;  
  
end Check_RI;
```

### DISCUSSION

The solution is obviously preferable. It is easier to read and, therefore, will be easier to maintain.

Remember, there must be an alternative specified for every possible value of type Character. In the solution, use of the alternative when others is an elegant way of providing for all choices other than R, U, or Y.

Use of the bar (|) in the solution allows multiple values for RI\_Field, e.g., R, U, or Y, to cause the same sequence of statements to be executed. More compact code is created by writing

```
when 'R' | 'U' | 'Y' =>  
    Status := OK;  
    Route_Message;
```

than by writing

```
when 'R' =>  
    Status := OK;  
    Route_Message;  
when 'U' =>  
    Status := OK;  
    Route_Message;  
when 'Y' =>  
    Status := OK;  
    Route_Message;
```

as in the alternative solution.

The choices R, U, and Y may not be expressed as a discrete range because they are not consecutive values. A through Q may be expressed as a discrete range as they are consecutive values.



This page intentionally left blank

## EXERCISE 5.3

### OBJECTIVE

To illustrate Boolean types.

### TUTORIAL

The type Boolean in Ada is a predefined type. It is implemented as an enumeration type with enumeration literals True and False as follows:

```
type Boolean is (False, True);
```

The format for declaration of a variable of type Boolean is as follows:

```
Variable_Name : Boolean;
```

The user-supplied name for the object, Variable\_Name is followed by the required colon, which is then directly followed by the type, Boolean. The semicolon is the required statement terminator. As a result of the declaration, the object, Variable\_Name, can take on one of the values True or False and only one of those values. Operations on Variable\_Name as well as on any objects of type Boolean, are restricted to those shown in Table 5-2. The logical operators are defined in Table 5-3.

Since type Boolean is an enumeration type, its values are ordered. Specifically, False < True is True, while False > True is False.

As an example, consider the following declarations for Sensor\_1\_Enabled and Sensor\_2\_Enabled;

```
Sensor_1_Enabled : Boolean;
```

Sensor\_2\_Enabled could be initialized as in

```
Sensor_2_Enabled : Boolean := False;
```

Notice how Sensor\_2\_Enabled is initialized to indicate a disabled state.

TABLE 5-2  
BOOLEAN OPERATORS

OPERATORS	RESULT TYPE
<u>Relational</u>	
=   /=   <   <=   >   >=	Boolean
<u>Logical</u>	
not   and   or   xor	Boolean

TABLE 5-3  
LOGICAL OPERATORS

<table><tr><td>and</td><td>T</td><td>F</td></tr><tr><td>T</td><td>T</td><td>F</td></tr><tr><td>F</td><td>F</td><td>F</td></tr></table> <p>and : True only if both operands True</p>	and	T	F	T	T	F	F	F	F	<table><tr><td>or</td><td>T</td><td>F</td></tr><tr><td>T</td><td>T</td><td>T</td></tr><tr><td>F</td><td>T</td><td>F</td></tr></table> <p>or : True if any operand True</p>	or	T	F	T	T	T	F	T	F
and	T	F																	
T	T	F																	
F	F	F																	
or	T	F																	
T	T	T																	
F	T	F																	
<table><tr><td>xor</td><td>T</td><td>F</td></tr><tr><td>T</td><td>F</td><td>T</td></tr><tr><td>F</td><td>T</td><td>F</td></tr></table> <p>xor : True only if one operand True</p>	xor	T	F	T	F	T	F	T	F	<table><tr><td>not</td><td>T</td><td>F</td></tr><tr><td></td><td>F</td><td>T</td></tr></table> <p>not : Negation, reverses value</p>	not	T	F		F	T			
xor	T	F																	
T	F	T																	
F	T	F																	
not	T	F																	
	F	T																	

It may be assigned to as in

```
Sensor_1_Enabled := True;
```

to change its state and it may be compared as in

```
Sensor_1_Enabled = True;
```

to determine its state.

The benefit of having Boolean objects is that logical expressions may be formed. Consider the declarations

```
Sensor_1_Enabled, Sensor_2_Enabled : Boolean;
```

which would, therefore, allow the expression

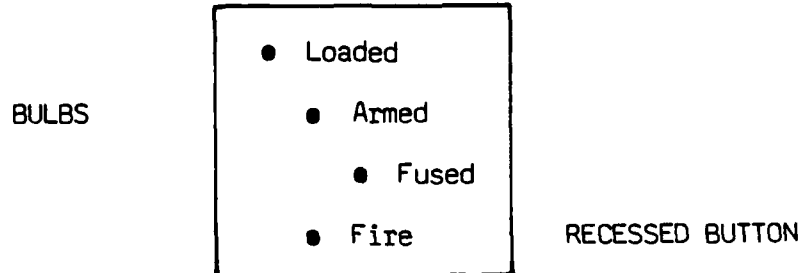
```
Sensor_1_Enabled and Sensor_2_Enabled
```

to be tested and statements executed only if both sensors were enabled.

This page intentionally left blank

### PROBLEM

The following diagram represents a simplified version of a missile control panel.



The control panel consists of three bulbs which can take on either the color red or green. Red indicates that the indicated activity has not occurred, green indicates that the indicated activity has occurred. Additionally, the control panel has one recessed button used to fire the missile.

For this exercise, declare and initialize three objects of type Boolean to represent the three bulbs.

This page intentionally left blank

## SOLUTION TO EXERCISE 5.3

### SOLUTION

The three object declarations and their appropriate initializations are as follows:

```
Loaded : Boolean := False;  
Armed  : Boolean := False;  
Fused  : Boolean := False;
```

### RATIONALE FOR THE SOLUTION

The object declarations have the format specified in the tutorial portion of the exercise. The name, e.g., Loaded, is followed by the colon, which is directly followed by the type, Boolean, which is then directly followed by the required initializations and the terminating semicolon.

Upon initialization we want to initialize the three objects to False since all the corresponding lamps are off.

### ALTERNATIVE SOLUTION

```
Loaded, Armed, Fused : Boolean := False;
```

### DISCUSSION

This statement declares three separate objects, Loaded, Armed, and Fused, each of type Boolean and initialized to False. This kind of statement is used when declaring several objects of the same type. It is essentially the same as the first solution.

By declaring the three bulbs as Boolean objects, we can form Boolean expressions such as

```
Loaded and Armed and Fused
```

which can be used to test the current state of the missile.

If the instructions for the problem had not specified Boolean objects, you might have been tempted to do the following:



```
type Panel_Bulb is (Loaded, Armed, Fused);
```

By declaring type Panel\_Bulb as an enumeration type with values Loaded, Armed, and Fused, the capability to form logical expressions is lost. The only operations allowed on objects of type Panel\_Bulb are relational or membership operations (See Table 5-1).

A similar discussion would surround the following declarations:

```
type Loaded is (Red, Green);  
type Armed  is (Red, Green);  
type Fused  is (Red, Green);
```

## EXERCISE 5.4

### OBJECTIVE

To illustrate the if statement

### TUTORIAL

The if statement allows selection of a sequence of statements depending on the value of a Boolean expression. There are two basic forms of the if statement. Each is described below. Condition must be a Boolean expression and, therefore, evaluates to either True or False.

#### Form 1

```
if Condition then
    statement(s);
end if;
```

If Condition evaluates to True, the statement(s) enclosed by the if statement are executed, the if statement terminates and control passes to the first statement after the if statement. If Condition evaluates to False, the statement(s) enclosed within the if statement are not executed, the if statement terminates, and control passes to the first statement after the if statement.

#### Example Form 1

The following

```
if not Sensor_1 then
    Enable_Sensor;
end if;
```

will ensure that Sensor\_1 is enabled before any statements after the if statement are executed.

## Form 2

```
if Condition_1 then
    statement(s)_1;
elsif Condition_2 then
    statement(s)_2;

-- perhaps other elsif alternatives

end if;
```

If Condition\_1 evaluates to True, then statement(s)\_1 is executed, the if statement terminates and control passes to the first statement after the if statement. If Condition\_1 is not True, then Condition\_2 is evaluated. If Condition\_2 evaluates to True, then statement(s)\_2 is executed, the if statement terminates and control passes to the first statement after the if statement. If Condition\_2 evaluates to False, then the next condition is evaluated, and so on. If no Condition\_i evaluates to True, the if statement terminates and control passes to the first statement after the if statement.

## Example Form 2

The following sets an object Largest to the largest of X and Y and prints a message if they are equal.

```
if X > Y then
    Largest := X;
elsif X < Y then
    Largest := Y;
elsif X = Y then
    Put ("They are equal");
end if;
```

An else alternative may be added to each of these two basic forms. Table 5-4 depicts the resulting structures. The addition of the else alternative provides an opportunity for certain statements to be executed if the Condition in Form 1, or any of the Condition\_i in Form 2 do

TABLE 5-4  
BASIC IF STATEMENT FORMS

<p><u>Form 1(a)</u></p> <pre> if <u>Condition</u> then   <u>statement(s)</u>; end if; </pre>	<p><u>Form 1(b)</u></p> <pre> if <u>Condition</u> then   <u>statement(s)</u>; else   <u>else statement(s)</u>; end if; </pre>
<p><u>Form 2(a)</u></p> <pre> if <u>Condition 1</u> then   <u>statement(s) 1</u>; elseif <u>Condition 2</u> then   <u>statement(s) 2</u>;  -- possibly other elsifs  end if; </pre>	<p><u>Form 2(b)</u></p> <pre> if <u>Condition 1</u> then   <u>statement(s) 1</u>; elseif <u>Condition 2</u> then   <u>statement(s) 2</u>;  -- possibly other elsifs  else   <u>else statement(s)</u>; end if; </pre>

not evaluate to True. In those cases, the statements following the "else" are executed. Finally, there must be only one else alternative in an if statement and when appearing, it must be the last alternative.

If statements may be nested, however proper use of the "elsif" alternative eliminates the need for nested if statements in certain instances. For example, the floating point object Discriminant represents the familiar discriminant of quadratic equations,  $b^2 - 4ac$ . The following

```
if Discriminant < 0.0 then
    Process_Complex_Roots;
else
    if Discriminant = 0.0 then
        Process_Two_Real_Roots;
    else
        Process_Two_Distinct_Roots;
    end if;
end if;
```

becomes more readable and requires fewer end ifs when coded as

```
if Discriminant < 0.0 then
    Process_Complex_Roots;
elsif Discriminant = 0.0 then
    Process_Two_Real_Roots;
else
    Process_Two_Distinct_Roots;
end if;
```

PROBLEM

Recode the problem of EXERCISE 5.2 using an if statement.

This page intentionally left blank

## SOLUTION TO EXERCISE 5.4

### SOLUTION

```
procedure Check_RI (RI_Field : in Character;
                    Status    : out Status_Type) is

    procedure Route_Message is separate;
    procedure Error_Routine is separate;

begin -- Check_RI

    if RI_Field = 'R' or RI_Field = 'U' or RI_Field = 'Y' then
        Status := OK;
        Route_Message;
    else
        Status := Error;
        Error_Routine;
    end if;

end Check_RI;
```

### RATIONALE FOR THE SOLUTION

The Condition in the if statement

RI\_Field = 'R' or RI\_Field = 'U' or RI\_Field = 'Y'

is a compound Boolean expression. If any one of the simple Boolean expressions

RI\_Field = 'R'  
RI\_Field = 'U'  
RI\_Field = 'Y'

evaluates to True, then the entire compound Boolean expression evaluates to True. This follows from the definition of the logical operator or defined in Table 5-3. If the compound Boolean expression evaluates to True, then Status is assigned the value OK and Route\_Message is called.



If RI\_Field has an illegal value which is neither of the values R, U, or Y, then Status is assigned the value Error and Error\_Routine is called.

#### ALTERNATIVE SOLUTION

```
procedure Check_RI (RI_Field : in Character;
                    Status    : out Status_Type) is

    procedure Route_Message is separate;
    procedure Error_Routine is separate;

begin -- Check_RI

    if RI_Field = 'R' then
        Status := OK;
        Route_Message;
    elsif RI_Field = 'U' then
        Status := OK;
        Route_Message;
    elsif RI_Field = 'Y' then
        Status := OK;
        Route_Message;
    else
        Status := Error;
        Error_Routine;
    end if;

end Check_RI;
```

#### DISCUSSION

The alternative solution duplicates code unnecessarily. The code as presented in the solution is preferable.

## **Section 6**

### **NUMERIC TYPES**

**6.1 User Defined Integer Types/Objects**

**6.2 User Defined Real Types/Objects**

**6.3 Subtypes**

This page intentionally left blank

## EXERCISE 6.1

### OBJECTIVE

To demonstrate integer types and objects.

### TUTORIAL

Integer and real types are used for numeric calculations. An integer type is defined by the range of possible values that can be assigned to an object of that type.

The format for an integer type declaration is:

```
type Integer_Type_Name is range Lower .. Upper;
```

Integer\_Type\_Name is the identifier for the type. The range Lower .. Upper is called the range constraint. It specifies that an object of type Integer\_Type\_Name may only have values X in the range Lower <= X <= Upper.

Notice that the word "integer" is not included in an integer type declaration. The word "integer" refers to a predefined type whose range is machine-dependent (as explained in Exercise 4.1). Integer, as used elsewhere in this text, refers to the class of integer types.

A declared integer type is distinct from any other integer type. Available operators are numeric, relational, membership, and unary. (Refer to Exercise 4.1 for elaboration of integer operators).

The following type declaration

```
type Small_Integer_Type is range 1 .. 10;
```

creates a type named Small\_Integer\_Type. Objects of this type may only take on values between one and ten, inclusive.

The range constraint cannot be excluded in an integer type declaration. It always has the following format:

```
range Lower .. Upper
```

The bounds, Lower and Upper, must be static expressions (i.e. their value must be known at compile time) of some integer type.

Here are some other examples of integer type declarations:

```
type Submarine_Count_Type is range 0 .. 250;  
type Miles_Flown is range 0 .. 10_000;  
type Size_Is range 1 .. 38;  
type Altitude_Above_Sea_Level is range -5E2 .. 1E4;
```

Once an integer type has been declared, objects of that type may be created. The format for declaring an object is:

Variable\_Name : Integer\_Type\_Name;

where Variable\_Name is the identifier for the object, and Integer\_Type\_Name is the name of a declared integer type. For example, given the type declarations shown above, a variable can be created to keep a count of the number of submarines in the Mediterranean as follows:

```
Submarines_in_Mediterranean : Submarine_Count_Type;
```

Separate variables to keep track of submarines in the Barents Sea, the Bering Sea, and the Indian Ocean can also be declared:

```
Submarines_in_Barents,  
Submarines_in_Bering,  
Submarines_in_Indian : Submarine_Count_Type;
```

These variables are of type Submarine\_Count\_Type, and all of the integer operations can be applied to them.

By contrast, if we declare a variable as shown below:

```
Miles_This_Month : Miles_Flown;
```

the following constructs would all be illegal:

```
Miles_This_Month := Submarines_in_Bering;  
Miles_This_Month > Submarines_in_Indian  
Miles_This_Month + Submarines_in_Barents < 3
```

Miles\_Flown and Submarine\_Count\_Type are distinct types and cannot be freely mixed.

There are two reasons for defining new integer types, rather than using the predefined type integer. One is to take advantage of the built-in type checking mechanism to prevent meaningless operations, like comparing numbers of submarines to miles flown. The other reason is machine independence. Type declarations allow the compiler to choose the most appropriate representation for that type's values on the target machine.

When an object of an integer type is created its implicit initial value is undefined. However, the programmer may specifically assign an initial value in the declaration as follows:

Variable\_Name : Integer\_Type\_Name := Initial\_Value;

Initial\_Value must be an expression of type Integer\_Type\_Name, and must satisfy the corresponding range constraint.

As an example, suppose there are initially two submarines in Baffin Bay. The statement

Submarines\_in\_Baffin : Submarine\_Count\_Type := 2;

would create the variable Submarines\_in\_Baffin and assign to it the value two. Its value may later be changed by assignment or when updated by a procedure.

A constant declaration creates an object of an integer type whose value does not change as follows:

Constant\_Name : constant Integer\_Type\_Name := Value;

Constant\_Name is declared a constant of Integer\_Type\_Name by using the reserved word "constant." Initialization must be included, but need not be static. For example, a constant declared in a subprogram could be dependent on the parameter values with which the subprogram is called.

A constant declaration would be suitable to describe the number of submarines that the U.S. built in 1961,

Submarines\_Built\_1961 : constant Submarine\_Count\_Type := 28;

because the number of submarines built that year is not variable. Once declared, the value of Submarines\_Built\_1961 cannot be modified.

This page intentionally left blank

PROBLEM

An airport in Montana monitors the weather in the surrounding area. Write the integer type declaration and object declarations for the temperature to the nearest whole degree (in Fahrenheit) in three nearby cities, Helena, Billings, and Great Falls. Also declare an object for the freezing temperature, 32 degrees.



AD-A165 345

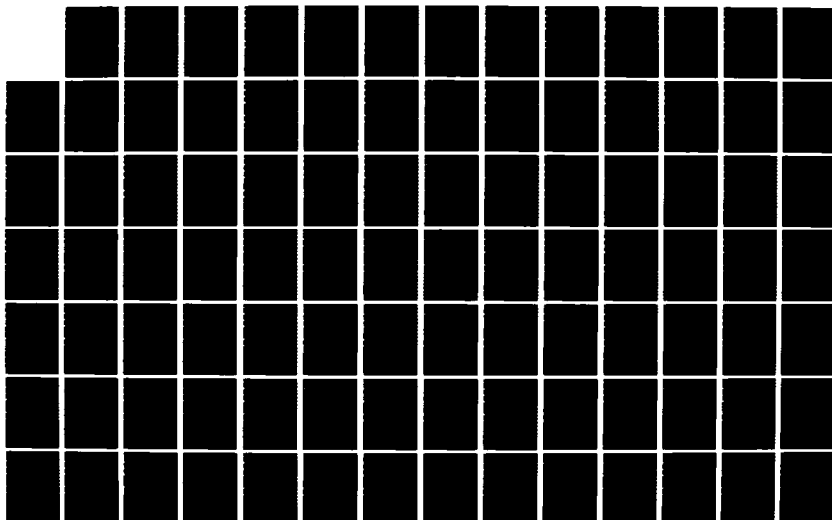
ADA (TRADEMARK) PRIMER(U) SOFTECH INC WALTHAM MA 1986  
DADB07-83-C-K506

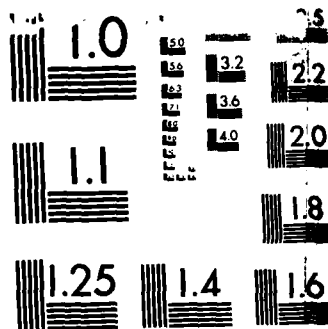
3/5

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A

This page intentionally left blank

## SOLUTION TO EXERCISE 6.1

### SOLUTION

```
type Temperature_Type_F is range -40 .. 120;

Temperature_Helena,
Temperature_Billings,
Temperature_Great_Falls : Temperature_Type_F;

Freezing_Temperature : constant Temperature_Type_F := 32;
```

### RATIONALE FOR THE SOLUTION

The type declaration for `Temperature_Type_F` establishes a type for temperature values in Fahrenheit. The range of allowed temperatures is -40 degrees to 120 degrees, appropriate for Montana.

The objects declared, `Temperature_Helena`, `Temperature_Billings`, and `Temperature_Great_Falls`, are each of type `Temperature_Type_F`, and accordingly take temperature values in the specified range.

`Freezing_Temperature` is declared a constant object because its value does not change.

Integer types and objects are appropriate for temperature calculations because temperature values are integers, and because arithmetic operations can be performed. For example, it may be desirable to subtract the temperature readings of distant towns to determine the temperature difference. Or, the temperature readings may be added and divided to determine the average temperature in the state.

### ALTERNATIVE SOLUTION

```
Temperature_Helena,
Temperature_Billings,
Temperature_Great_Falls : Integer range -40 .. 120;

Freezing_Temperature : Integer := 32;
```

## DISCUSSION

(This type of integer declaration is described in Exercise 4.1.)  
Temperature\_Helena, Temperature\_Billings, and Temperature\_Great\_Falls are declared separate objects of the predefined type integer, accepting integer values between -40 and 120. However, by declaring a type specifically for temperature values, as in the first solution, the programmer allows changes in the program to be made more easily, insures type matching, and improves readability. If it were desirable to add other towns in Montana to the program, a separate temperature type insures that all of these objects are of the same type. Or, if a change were made to the range of temperatures, the temperature type alone could be modified.

Freezing\_Temperature in the alternate solution is declared an object of predefined type integer, initialized to 32. However, it is not a constant, and therefore may be modified in the course of program calculations. Making Freezing\_Temperature a constant prevents its value from being unintentionally changed.

## EXERCISE 6.2

### OBJECTIVE

To demonstrate real types and objects.

### TUTORIAL

For numeric calculations requiring a decimal part Ada provides two types, floating point and fixed point. Their use is similar to real values in other languages.

Unlike integer types, neither floating point nor fixed point are exact representations. Floating point values have an error relative to the size of the number, and fixed point values have a degree of error that is fixed.

There exists a predefined type for floating point called Float, whose specifications are implementation dependent (refer to Exercise 4.3). User-defined floating point types have the following declaration format:

```
type Float_Type_Name is digits D;
```

"Digits" is a reserved word, and the value D is provided by the user. It specifies the relative precision of Float\_Type\_Name. D corresponds to the minimum number of significant decimal digits. It must be a static expression of some integer type having a positive (nonzero) value.

Specifically, the declaration

```
type Mass_Type is digits 5;
```

defines a floating point type called Mass\_Type. Objects of this type have real values with a precision of at least five significant decimal places.

Floating point type declarations may also include a range constraint. The format is then:

```
type Float_Type_Name is digits D range Lower .. Upper;
```

In the following type declaration,

```
type Sine_Type is digits 10 range -1.0 .. 1.0;
```

Sine\_Type accomodates real numbers between -1.0 and 1.0, inclusive, having an accuracy of at least 10 decimal digits.

Fixed point type is somewhat similar to floating point, though used less frequently. A type declaration for fixed point has the following format:

```
type Fixed_Type is delta D range Lower .. Upper;
```

The reserved word "delta" and the value "D" correspond to the accuracy of the type. D must be a static expression of some real type with a positive (nonzero) value.

The declaration,

```
type Current_Type is delta 0.250 range 0.0 .. 500.0;
```

defines a fixed point type, Current\_Type, with an accuracy of 0.250 and a range of values between 0.0 and 500.0.

The range constraint, Lower .. Upper, is optional in a floating point type declaration, but it must be included in a fixed point type declaration. The range bounds, Lower and Upper, in both floating and fixed point types must be static expressions of some real type. The allowable operations for both types are numeric, relational, and unary (as explained in Exercise 4.3).

Here are some examples of floating point and fixed point type declarations:

```
type Real_Type_Fl is digits 8;  
type Real_Type_Fx is delta 0.125 range 1.0E-4 .. 1.0E4;  
type pH_Type is digits 4 range 0.0 .. 14.0;  
type Acceleration_Type is delta 0.125 range -150.0 .. Max_Acc;
```

Max\_Acc above must be previously declared and evaluate to a real value. Notice that Real\_Type\_Fx and Acceleration\_Type have the same delta, 0.125. It may be preferable to declare the value of delta or digits separately as in:

```
Digits_Value : constant := 5;  
Delta_Value : constant := Plane_Model * 1.0E-3;
```

Digits\_Value and Delta\_Value are of an unnamed universal type. Digits\_Value may now be used in place of 5. Plane\_Model must be static and positive.

Several types with these values can then be declared:

```
type Newtons_Type is digits Digits_Value;  
type Cosine_Type is digits Digits_Value;  
type Hemoglobin_Count is digits Digits_Value range 5.0 .. 15.0;  
type Joules_Type is delta Delta_Value range 0.0 .. 1000.0;  
type Average_Velocity_Type is delta Delta_Value range 0.0 .. Max_Vel;
```

Note that although Newtons\_Type and Cosine\_Type have the same specifications, they are distinct types and cannot be mixed. One other reminder -- the digits value is always an integer and the delta value is always real. An integer literal may not have a decimal point, and a real literal must.

The declaration of objects of floating point or fixed point types is similar to that of integer types. The basic format is:

Real\_Variable\_Name : Real\_Type\_Name;

where Real\_Variable\_Name is the identifier for the object and Real\_Type\_Name is the name of a declared real type. For example, given the type declarations above for Newtons\_Type and Joules\_Type, objects for values measured in newtons or joules can be created:

```
Weight_of_Pack      : Newtons_Type;  
Weight_of_Truck     : Newtons_Type;  
Exothermic_Energy   : Joules_Type;
```

Since Weight\_of\_Pack and Weight\_of\_Truck are of the same type, they may be compared or assigned to one another using real operators. For example,

```
Weight_of_Truck := Weight_of_Truck + 50.0 * Weight_of_Pack;
```

However, neither Weight\_of\_Pack nor Weight\_of\_Truck may be compared to Exothermic\_Energy because the types do not match.

An object of type real may be initialized in the declaration using the following format:

Real\_Variable\_Name : Real\_Type\_Name := Initial\_Real\_Value;

Initial\_Real\_Value must be an expression of type Real\_Type\_Name, and must satisfy any corresponding range constraint.



Suppose that a reaction labeled A yields 203.449 joules of energy during the first minute when a certain catalyst is used. An object for that value may be created as follows:

```
Reaction_A_Minute_1 : Joules_Type := 203.449;
```

Reaction\_A\_Minute\_1 is declared an object for the value of the energy in joules given off during the first minute of Reaction A. It is of type Joules\_Type, having an initial value for the first catalyst of 203.449.

A real object having a constant value may be declared as follows:

```
Real_Constant_Name : constant Real_Type_Name := Initial_Real_Value;
```

where "constant" is a reserved word. Initial\_Real\_Value must be included, but need not be static.

If Reaction A always gives off 889.030 total joules of energy, a constant declaration would be appropriate:

```
Reaction_A_Total : constant Joules_Type := 889.030;
```

The value of Reaction\_A\_Total cannot then be changed.

### PROBLEM

A real-time information system requires continually updated information regarding the Richter scale readings of the earthquakes in the North Pacific off the coast of Japan. Also to be reported is the exact distance in nautical miles between the epicenter of the earthquake and each of three ships in that area, the U.S.S. Enterprise, the U.S.S. Wasp, and the U.S.S. Kennedy. There is initially no earthquake. (Richter scale values are between 0.0 and 10.0.) Write the requisite floating point declarations using any number of decimal digits you think appropriate.

This page intentionally left blank

## SOLUTION TO EXERCISE 6.2

### SOLUTION

```
type Richter_Type is digits 4 range 0.0 .. 10.0;  
type Nautical_Miles_Type is digits 3;
```

```
Earthquake_Reading      : Richter_Type;  
Distance_To_Enterprise,  
Distance_To_Wasp,  
Distance_To_Kennedy     : Nautical_Miles_Type := 0.0;
```

### RATIONALE FOR SOLUTION

Richter\_Type is created for Richter scale values. It is declared a floating point type with four decimal digits and the appropriate range of possible values between 0.0 and 10.0. Nautical Miles Type is created for nautical mile values. It is declared with three decimal digits. No range is specified because the maximum distance between an earthquake and a ship is unknown.

Earthquake\_Reading is created to store the Richter scale reading of the current earthquake. It is, naturally, of type Richter\_Type. The distances between the earthquake and the three ships are measured in nautical miles, and are therefore of type Nautical\_Miles\_Type. Three separate objects are declared for each distinct distance.

Floating point types are appropriate for the types in this problem because the values involved are real. Fixed point could also have been used.

### ALTERNATIVE SOLUTION

```
Earthquake_Reading      : Float range 0.0 .. 10.0;  
Distance_To_Enterprise,  
Distance_To_Wasp,  
Distance_To_Kennedy     : Float := 0.0;
```

## DISCUSSION

(Predefined float is explained in Exercise 4.3.)

In the alternate solution Earthquake Reading is declared a variable for real values. Distance\_To\_Enterprise, Distance\_To\_Wasp, and Distance\_To\_Kennedy are also declared separate real variables. All are defined by the implementation-dependent specifications of predefined float. It is preferable, however, to declare separate types for Richter readings and nautical mile distances, as in the first solution, because the value of digits can be specified and known. The program is then independent of the implementation. Also, changes in the specifications of those types can readily be made, and additional ship or earthquake variables can easily be added. By declaring specific types for Richter scale readings and nautical miles the programmer takes advantage of type checking mechanisms, and enhances readability and maintainability.

## EXERCISE 6.3

### OBJECTIVE

To demonstrate subtypes.

### TUTORIAL

A subtype is a subset of some type. The original type from which the subset is taken is called the base type. For example,

```
subtype Upper_Case_Letter_Subtype is Character range 'A' .. 'Z';
```

declares a subtype called Upper\_Case\_Letter\_Subtype which is a subset of the predefined type Character.

A subtype does not introduce a distinct type. It is merely an expression for a constrained version of the existing base type. Any type, including predefined types, may be a base type for a subtype declaration.

The constraint defining the subtype applies to the values of the base type, never to the applicable operations.

The format for a subtype declaration is:

```
subtype Subtype_Name is Base_Type [Constraint];
```

Subtype\_Name is the identifier for the new subtype. Base\_Type is the base type of which Subtype\_Name is a subset. Constraint may or may not be included (as is suggested by the brackets). If a constraint is specified it restricts the possible values for Subtype\_Name. If it is left out the allowed values for Subtype\_Name include all of the allowed values for the objects of Base\_Type. The form of Constraint varies depending upon the category of Base\_Type. Its specifications cannot extend beyond the range of Base\_Type. Subtype\_Name is not a distinct type. It defines a subset of Base\_Type.

For example, given an ordinary integer type declaration,

```
type Integer_Base_Type is range 1 .. 100;
```

the following subtype declarations are valid:

```
subtype One_To_Fifty_Subtype is Integer_Base_Type range 1 .. 50;
subtype Forties_Subtype      is Integer_Base_Type range 40 .. 49;
subtype Eighties_Subtype     is Integer_Base_Type range 80 .. 89;
subtype One_To_One_Hundred_Subtype is Integer_Base_Type;
```

One\_to\_Fifty\_Subtype, Forties\_Subtype, Eighties\_Subtype, and One\_to\_One\_Hundred\_Subtype are all subtypes of Integer\_Base\_Type. One\_to\_Fifty\_Subtype allows only values  $X$  in the range  $1 \leq X \leq 50$ , Forties\_Subtype allows only values  $40 \leq X \leq 49$ , etc. One\_to\_One\_Hundred\_Subtype allows all values of Integer\_Base\_Type, because it is not further constrained. It is essentially another name for Integer\_Base\_Type. The following declaration,

```
subtype Zero_to_Five_Subtype is Integer_Base_Type range 0 .. 5;
```

is NOT allowed because the range specified for Zero\_to\_Five\_Subtype is not within the range of Integer\_Base\_Type.

Objects can be created with subtypes exactly as they are with types. The format is:

```
Variable_Name : Subtype_Name;
```

where Variable\_Name is the name of the new variable, and Subtype\_Name is the name of the subtype for Variable\_Name. Variable\_Name is of type Base\_Type, the parent of Subtype\_Name, but its values are constrained by the specifications of Subtype\_Name.

Subtype variables may also be initialized just as objects for types are:

```
Variable_Name : Subtype_Name := Initial_Value;
```

Initial\_Value must be of type Base\_Type, and must satisfy the stipulated constraint of Subtype\_Name.

For example, one can declare objects for the subtypes above as shown:

```
Integer_Base      : Integer_Base_Type      := 2;
One_to_Fifty      : One_to_Fifty_Subtype   := 42;
Forties           : Forties_Subtype        := 49;
Eighties          : Eighties_Subtype;
One_to_One_Hundred : One_to_One_Hundred_Subtype;
```

Because subtypes do not form a distinct type, the values of variables of subtypes and their base types may be assigned to one another.

```
Base_Type_Variable := Subtype_Variable;
```

is always legal. On the other hand,

```
Subtype_Variable := Base_Type_Variable;
```

is legal but potentially fatal if the value of Base\_Type\_Variable is within the range specified for Subtype\_Variable. So,

```
Integer_Base := Forties;
```

is always allowed. And,

```
One_To_Fifty := Integer_Base;
```

is also allowed, but only because the current value of Integer\_Base is within the range of One\_To\_Fifty. If we assign to Integer\_Base the value 75,

```
Integer_Base := 75;
```

the following statement would give rise to a run-time error:

```
One_To_Fifty := Integer_Base;
```

Finally, both

```
One_To_One_Hundred := Integer_Base;  
Integer_Base := One_To_One_Hundred;
```

are always allowed, because their types are different only in name.

The membership operators, "in" and "not in", check whether a value of a base type obeys the constraint of a subtype. For example, given the following type and subtype declarations:

```
type Security_Classification_Type is  
  (Unclassified, Confidential, Secret, Top_Secret);  
subtype Classified_Subtype is Security_Classification_Type  
  range Confidential .. Top_Secret;
```

```
Document_X : Classified_Subtype := Secret;
```

then

```
Document_X in Classified_Subtype
```

is True, because its current value, Secret, is within the range of Classified\_Subtype. And

```
Document_X not in Classified_Subtype
```

is False for the same reason.



Comparisons or assignments between variables of subtypes of the same base type succeed if their ranges overlap and if the current values of the variables satisfy those constraints.

```
One_To_Fifty := Forties;
```

is always allowed.

```
Forties := One_To_Fifty;
```

fails unless the current value of One\_To\_Fifty is within the range of Forties. Neither

```
Forties := Eighties;
```

nor

```
Eighties := Forties;
```

could possibly execute successfully, since the value of Eighties can never be in the range required for Forties, and vice versa.

It is also permissible to apply a constraint to a subtype to obtain a subtype of the same base type. For example, given the enumeration type declaration,

```
type Alphabet is (A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z);
```

one can create the following subtypes of subtypes:

```
subtype To_T is Alphabet range A .. T;  
subtype To_Q1 is To_T range A .. Q;  
subtype To_Q2 is To_Q1;  
subtype To_H is To_Q2 range A .. H;  
subtype E is To_H range E .. E;
```

Here are some examples of subtype declarations for several categories of types:

Predefined integer:

```
subtype Citizens_Subtype is Integer;  
subtype Age_Subtype is Integer range 0 .. 100;
```

Predefined float:

```
subtype X_Component_Subtype is Float range -50.0 .. 50.0;
```

Floating point:

```
type Density_Type is digits 8 range 1.0E-6 .. 10.0;
subtype Approximate_Density_Subtype is Density_Type digits 3;
subtype Low_Density_Subtype is Density_Type range 1.0E-6 .. 2.0;
subtype Gaseous_Density_Subtype is Density_Type digits 5
range 1.0E-6 .. 1.0;
```

Notice that the subtype constraint for a real type can apply to either the value, the accuracy, or both.

Fixed Point:

```
type Length_in_Cm_Type is delta 0.125 range 0.0 .. 100.0;
subtype Short_Length_Subtype is Length_in_Cm_Type range 0.0 .. 10.0;
```

Subtypes are declared primarily when the programmer knows what range of values will occur for a given type. By declaring a subtype he or she enables programming errors to be detected more easily by preventing incorrect value assignments. However, it is not necessary to create a subtype only for a constraint.

This page intentionally left blank

PROBLEM

An instrument measures the wavelengths of the light spectrum between 10 and 900 nanometers ( $1\text{E}-9$  meters). Within that range, ultraviolet light has wavelengths between 10 and 390 nm., visible light between 390 and 760 nm., and infrared between 760 and 900 nm. The wavelength of visible red light is between 630 and 635 nm. Write the requisite type, subtype, and object declarations to accomodate each of these categories of light.

This page intentionally left blank

## SOLUTION FOR EXERCISE 6.3

### SOLUTION

```
type Light_Spectrum_Type is range 10 .. 900;

subtype Ultraviolet_Subtype is Light_Spectrum_Type range 10 .. 390;
subtype Visible_Subtype    is Light_Spectrum_Type range 390 .. 760;
subtype Infrared_Subtype   is Light_Spectrum_Type range 760 .. 900;
subtype Visible_Red_Subtype is Visible_Subtype     range 630 .. 635;

Wavelength : Light_Spectrum_Type;
Ultraviolet : Ultraviolet_Subtype;
Visible     : Visible_Subtype;
Infrared    : Infrared_Subtype;
Visible_Red : Visible_Red_Subtype;
```

### RATIONALE FOR SOLUTION

Light\_Spectrum\_Type is declared an integer type with the appropriate range for light wavelengths from 10 to 900 nanometers. The floating point declaration,

```
type Light_Spectrum_Type is digits 5 range 1.0E-7 .. 90.0E-7;
```

would also be suitable for Light\_Spectrum\_Type. (The bounds in the range constraints of Ultraviolet\_Subtype, Visible\_Subtype, Infrared\_Subtype, and Visible\_Red\_Subtype would have to be changed accordingly.)

The solution also declares Ultraviolet\_Subtype, Visible\_Subtype, and Infrared\_Subtype subtypes of Light\_Spectrum\_Type, with constraints corresponding to their respective ranges of wavelength values. Visible\_Red\_Subtype is declared a subtype of the subtype Visible\_Subtype, because it is a sub-category of visible light, just as visible light is a category of the whole light spectrum. Whether Visible\_Red\_Subtype is made a subtype of Visible\_Subtype or Light\_Spectrum\_Type is a matter of circumstance and personal preference.

Wavelength is declared an object of type Light\_Spectrum\_Type, having allowed values in the entire range between 10 and 900. The declared objects of the subtypes Ultraviolet, Visible, and Infrared are really of type Light\_Spectrum\_Type, but allowable values are restricted by the stipulations of the subtype declaration. Visible\_Red is also of type Light\_Spectrum\_Type, with allowed values in the range 630 to 635.

Subtypes can be used for the categories Ultraviolet, Visible, Infrared, and Visible\_Red in the solution because the programmer knows exactly what values each of these types of light can have. The declaration of subtypes prevents inappropriate assignments and makes the program easier to read and maintain. However, it must be stressed that the use of subtypes depends very much upon the personal preference of the programmer and the circumstances for which the types are declared. It is perfectly possible that the subtypes declared in the solution would be unnecessary for some program situation.

#### FIRST ALTERNATIVE SOLUTION

```
type Light_Spectrum_Type is range 10 .. 900;

Wavelength : Light_Spectrum_Type;
Ultraviolet : Light_Spectrum_Type;
Visible     : Light_Spectrum_Type;
Infrared    : Light_Spectrum_Type;
Visible_Red : Light_Spectrum_Type;
```

#### DISCUSSION

The alternate solution uses no subtypes. All of the objects are declared of the general type `Light_Spectrum_Type`. This solution is legal and apt, but does not prevent the assignment of inappropriate values to these objects. In fact, it is not clear exactly what wavelengths are allowed for visible light or any of the other kinds of light referred to in the problem. Specification of those ranges is very advantageous.

#### SECOND ALTERNATIVE SOLUTION

```
type Light_Spectrum_Type is range 10 .. 900;
type Ultraviolet_Type    is range 10 .. 390;
type Visible_Type        is range 390 .. 760;
type Infrared_Type       is range 760 .. 900;
type Visible_Red_Type    is range 630 .. 635;

Wavelength : Light_Spectrum_Type;
Ultraviolet : Ultraviolet_Type;
Visible     : Visible_Type;
Infrared    : Infrared_Type;
Visible_Red : Visible_Red_Type;
```

## DISCUSSION

The second alternate solution declares a separate type for each kind of light. The objects Wavelength, Ultraviolet, Visible, Infrared and Visible Red are subsequently declared of the corresponding type. Again, this solution is perfectly legal in all respects. Additionally, it takes advantage of type-checking mechanisms. However, it is unnecessary to create five types. All of these objects are for wavelength values. In the first solution, all of the objects could be compared to one another because they are of the same type. For example,

```
Wavelength := Infrared;
```

would be legal and obviously useful. But in the second alternate solution that assignment could not be used. The only apparent reason for the second alternate solution is to ensure that assignment values are correct. And this can be accomplished just as well with subtypes.



This page intentionally left blank

## **Section 7**

### **ADVANCED FEATURES OF SCALAR TYPES**

**7.1 Short Circuit Control Form**

**7.2 Case Statement with Numerics**

**7.3 Attributes**

This page intentionally left blank

## EXERCISE 7.1

### OBJECTIVE

To demonstrate the short circuit control forms.

### TUTORIAL

Ada does not specify the order in which the separate terms of a compound Boolean expression are evaluated. One cannot know, for the expression below,

First\_Boolean\_Term and Second\_Boolean\_Term

whether First\_Boolean\_Term or Second\_Boolean\_Term is evaluated first.

This is not ordinarily a problem, but consider the following statement:

if  $1/X > 3.5$  then ...

Evaluation of  $1/X$  will cause a runtime error (technically, it will raise an exception) if  $X=0$ . To prevent the error it is not sufficient to write

if  $X > 0.0$  and  $1/X > 3.5$  then ...

because both expressions must be evaluated to form a logical and.

For these situations Ada offers two short circuit operators, "and then" and "or else," which correspond to the logical operators "and" and "or," respectively, but are defined so that operands are evaluated from left to right, and only as far as necessary.

The use of the short circuit operator "and then" has the same result as "and." It additionally defines the order of evaluation in a compound Boolean expression to proceed from left to right. If the above expression were written:

First\_Boolean\_Term and then Second\_Boolean\_Term

First\_Boolean\_Term would necessarily be evaluated before Second\_Boolean\_Term; if First\_Boolean\_Term were found to be false, Second\_Boolean\_Term would not be evaluated at all.

Similarly, if the short circuit operator "or else," having the same result as "or," were employed as follows,

First\_Boolean\_Term or else Second\_Boolean\_Term

First\_Boolean\_Term would be evaluated first; Second\_Boolean\_Term would be evaluated only if First\_Boolean\_Term is false.

The advantage in using short circuit operators is that evaluation of the second term can be prevented. If the first term in an "and then" expression evaluates to false the entire expression must be false.

Consequently, the second term need not and will not be evaluated. In an "or else" expression, the second term will not be evaluated if the first evaluates to be true because the expression is then known to be true.

For example, given the declarations:

```
type Satellite_Count_Type is range 0 .. 250;
Total_Satellites          : Satellite_Count_Type := 40;
Communication_Satellites  : Satellite_Count_Type := 8;
Weather_Satellites        : Satellite_Count_Type := 0;
```

the expressions in the statement:

```
if (Communication_Satellites = Weather_Satellites) and then
    (Communication_Satellites < Total_Satellites) then
    .
end if;
```

are evaluated from left to right. The second term ( $\text{Communication\_Satellites} < \text{Total\_Satellites}$ ) is never evaluated because the first term ( $\text{Communication\_Satellites} = \text{Weather\_Satellites}$ ) is false. The entire expression must therefore, be false. There is not any need to evaluate a further expression and so it is not done.

In the statement:

```
if (Total_Satellites > Weather_Satellites) or else
    (Total_Satellites > Communication_Satellites) then
    .
end if;
```

the first expression ( $\text{Total\_Satellites} > \text{Weather\_Satellites}$ ) evaluates to true. Therefore, the whole expression must be true, and the second expression is not evaluated.

As previously indicated, short circuit control forms are mostly used when the evaluation of the second term of a Boolean expression may cause an error or exception.

### PROBLEM

In an evaluation of a damped harmonic oscillator the roots of the quadratic equation are required to find the oscillation omega. If omega is less than 3.0 a procedure called Process\_Specs\_for\_Short\_Omega

```
procedure Process_Specs_for_Short_Omega (A,B,C : in Float);
```

should be invoked. A function Omega

```
function Omega (A,B,C : Float) return Float;
```

returns the values of the roots of the quadratic equation. It may only be called if the roots are real, i.e. if the discriminant is non-negative. Another function Discriminant

```
function Discriminant (A,B,C : Float) return Float;
```

returns the value of the discriminant. Write a statement that calls Process\_Specs\_for\_Short\_Omega if omega is less than 3.0.

This page intentionally left blank

## SOLUTION FOR EXERCISE 7.1

### SOLUTION

```
if (Discriminant (X, Y, Z) >= 0.0) and then
  (Omega (X, Y, Z) < 3.0) then
  Process_Specs_for_Short_Omega (X, Y, Z);
end if;
```

### RATIONALE FOR SOLUTION

Process\_Specs\_for\_Short\_Omega is to be called if  $(\text{Omega}(X, Y, Z) < 3.0)$ . However, Omega can only be evaluated if the discriminant is non-negative.  $(\text{Discriminant}(X, Y, Z) \geq 0.0)$  is, therefore, the first condition of the "and then" statement.

If  $(\text{Discriminant}(X, Y, Z) \geq 0.0)$  is false then the roots are not real, Omega is not evaluated, and Process\_Specs\_for\_Short\_Omega is not called. If it is true, Omega can be and is evaluated. If Omega is less than 3.0 Process\_Specs\_for\_Short\_Omega is invoked. If Omega is not less than 3.0 Process\_Specs\_for\_Short\_Omega is not invoked, and control passes to the next statement.

The "and then" operator is appropriate for this problem because it neatly ascertains that the discriminant is non-negative before allowing the evaluation of Omega.

### ALTERNATIVE SOLUTION

```
if Discriminant (X, Y, Z) >= 0.0 then
  if Omega (X, Y, Z) < 3.0 then
    Process_Specs_for_Short_Omega (X, Y, Z);
  end if;
end if;
```

### DISCUSSION

The alternate solution uses a nested if statement to ensure that the discriminant is positive before evaluating Omega. If  $(\text{Discriminant}(X, Y, Z) \geq 0.0)$  is false control immediately passes to the next statement. If it is true the next if statement is evaluated. If Omega is not less than 3.0, control passes out of both if statements. If it is less than 3.0 Process\_Specs\_for\_Short\_Omega is invoked.



The alternate solution solves the problem legally and reasonably simply. But extra nesting should be avoided if another method can be used. The first solution is much easier to read.

Most importantly, the alternative approach works if the boolean expression appears in an if statement, but not in the case of a loop. Loops are explained in Exercise 8.3. For example, if the problem required something like,

```
while (Discriminant (X, Y, Z) >= 0.0) and then
      (Omega (X, Y, Z) < 3.0)
loop
.
.
end loop;
```

it would not be easy to solve without the "and then" operator.

## EXERCISE 7.2

### OBJECTIVE

To demonstrate case statements with numerics.

### TUTORIAL

The case statement selects a sequence of statements for execution from several exclusive alternatives based on the value of a discrete expression. (Refer to Exercise 5.2 for a full explanation of case statements.) Ada allows discrete numeric values to be used in case statements. In a case statement with numerics the discrete expression is of an integer type, and the corresponding alternatives are integer literals. The format is:

```
case Integer_Expression is
  when Integer =>
    Sequence of Statements;
  when Integer =>
    Sequence of Statements;
  .
  .
end case;
```

Integer\_Expression is a discrete expression that evaluates to an integer value. The case statement alternatives, labeled Integer here, are simple integers. And Sequence of Statements is one or several statements to be executed.

Neither floating point nor fixed point types can be used as the case expression or choice values in a case statement.

The following example of a case statement with numerics,

```
case Frequency_of_Signal is
  when 223400 =>
    Adjust_for_Channel_1;
  when 241900 =>
    Adjust_for_Channel_2;
  when 349800 =>
    Adjust_for_Channel_3;
  when 2055000 =>
    Adjust_for_Channel_4;
  when others =>
    No_Channel;
end case;
```

has an expression Frequency\_of\_Signal that evaluates to some integer value. When Frequency\_of\_Signal is 223400, Adjust\_for\_Channel\_1 is invoked. When it is 241900, Adjust\_for\_Channel\_2 is invoked, etc. If Frequency\_of\_Signal is none of these values, No\_Channel is called. All of the features of ordinary case statements illustrated in Exercise 5.2 are available when using integers. The same sequence of statements can be executed by several integer alternatives by separating the alternatives by a bar. For example,

```
case Communication_Path is
  when 1 =>
    Find_Alternate_Path;
  when 2 | 3 | 4 =>
    Verify_Path;
  when others =>
    Report_Error;
end case;
```

has a discrete expression Communication\_Path. When Communication\_Path is 1 Find\_Alternate\_Path is invoked. And when Communication\_Path is 2, or 3, or 4 Verify\_Path is invoked. If Communication\_Path is none of these values, an error procedure is called.

And, as with ordinary case statements, a range of alternatives can be specified. The above can also be written:

```
case Communication_Path is
  when 1 =>
    Find_Alternate_Path;
  when 2 .. 4 =>
    Verify_Path;
  when others =>
    Report_Error;
end case;
```

# PROBLEM

Write a case statement that will select a procedure call for the frequency of a particular type of electromagnetic wave. If the frequency of the wave is within the range of AM radio (5E5 - 15E5 Hz.) the procedure Radio should be invoked. If the wave frequency is either VHF television (55E6 - 21E7 Hz.) or UHF television (4E8 - 8E8 Hz.) the procedure Television should be invoked. And if the frequency is that of radar (3E9 Hz.) the procedure Radar should be invoked. If the frequency is none of these types the procedure No\_Classification should be called.

This page intentionally left blank

## SOLUTION FOR EXERCISE 7.2

### SOLUTION

```
case Wave_Frequency is
  when 5E5 .. 15E5 =>
    Radio;
  when 55E6 .. 21E7 | 4E8 .. 8E8 =>
    Television;
  when 3E9 =>
    Radar;
  when others =>
    No_Classification;
end case;
```

### RATIONALE FOR SOLUTION

The solution has the required format for a case statement with range bounds given in the alternative choices. Wave\_Frequency is the integer frequency number. When it has values between 5E5 and 15E5 the procedure Radio is invoked, and control passes to the next statement. If it has values between 55E6 and 21E7 or 4E8 and 8E8 the procedure Television is invoked. And if its value is 3E9 Radar is invoked. If Wave\_Frequency has a value that is not within any of these ranges No\_Classification is called.

The use of integer literals in the case statement is appropriate for this problem because the wave frequencies are integer values. The ranges of frequencies can most easily be described numerically.

### ALTERNATIVE SOLUTION

```
if Wave_Frequency > 5E5 and Wave_Frequency < 15E5 then
  Radio;
elsif (Wave_Frequency > 55E6 and Wave_Frequency < 21E7) or
      (Wave_Frequency > 4E8 and Wave_Frequency < 8E8) then
  Television;
elsif Wave_Frequency = 3E9 then
  Radar;
else
  No_Classification;
end if;
```

## DISCUSSION

The alternative solution uses an if statement to call the appropriate procedure for the range of integer frequencies. This solution is perfectly legal and effective, but the simplicity of the first solution makes for easier reading, and therefore maintenance. The case statement is well suited for this problem.

## EXERCISE 7.3

### OBJECTIVE

To introduce attributes.

### TUTORIAL

An attribute is a predefined operation which allows the programmer to refer to a certain property of a declared type or object. The format is:

Name'Attribute\_Identifier

where Name is the name of a type or object, and Attribute\_Identifier is the name of the predefined attribute. The attribute is always preceded by a prime, sometimes called tick. There are over forty predefined attributes, but only a few will be addressed here.

Given the following type,

```
type Inert_Element_Type is
  (Helium, Neon, Argon, Krypton, Xenon, Radon);
```

the attribute 'First applied to Inert\_Element\_Type,

```
Inert_Element_Type'First
```

will deliver the first element of Inert\_Element\_Type, Helium. Similarly, 'Last applied to Inert\_Element\_Type,

```
Inert_Element_Type'Last
```

is Radon.

If we declare an object,

```
Inert_Gas : Inert_Element_Type;
```

Inert\_Gas could be assigned a value by using an attribute, as follows:

```
Inert_Gas := Inert_Element_Type'Last;
```

Attributes allow programs to be more portable. They can reference program elements without including specific machine-dependent characteristics. Programs are for the same reason easier to change.



For example, without attributes,

`Inert_Element_Type'Last`

would have to be referred to as Radon. If the elements in the type declaration of `Inert_Element_Type` were altered, the expressions for its elements using attributes would be accurate without being changed.

Attributes can only be applied to certain specified types. For example, the attribute `'Delta` may only pertain to a fixed point type. It delivers the value of the delta specified for the type. Given a type,

`type Coulombs_Type is delta 0.250 range 0.0 .. 250.0;`

the expression

`Coulombs_Type'Delta`

is 0.250. Similarly `'Digits` returns the digits value specified for a floating point type. Given a type for electron volts,

`type eV_Type is digits 5;`

the expression,

`eV_Type'Digits`

is 5. `'Digits` may be useful, too, if the programmer wishes to refer to the digits value specified for predefined float on the machine being used.

`'Digits` and `'Delta` may only be applied to floating point and fixed point types, respectively. `'First` and `'Last`, however, may pertain to several types.

Given an integer type,

`type Volts_Type is range 0 .. 1000;`

the expression,

`Volts_Type'First`

returns 0, and

`Volts_Type'Last`

returns 1000.

'Base returns the base type, and must be a prefix to another attribute.  
If there are subtypes,

```
subtype Volts_In_Circuit_Subtype is Volts_Type range 0 .. 50;  
subtype eV_Subtype is eV_Type range 4.8E-10 .. 1E-8;
```

then the following expressions can be written:

```
Volts_In_Circuit_Subtype'Base'Last  
eV_Subtype'Base'Digits
```

Volts\_In\_Circuit\_Subtype'Base'Last returns the last value in the base type of Volts\_In\_Circuit\_Subtype, which is 1000. And eV\_Subtype'Base'Digits returns the digits value specified in the type declaration of the base type of eV\_Subtype, which is 5.

This page intentionally left blank

PROBLEM

Given a declared enumeration subtype called Gaseous\_Inert\_Element\_Subtype, write a boolean expression that determines whether the last element of its base type is in the subtype.

This page intentionally left blank

## SOLUTION FOR EXERCISE 7.3

### SOLUTION

```
Gaseous_Inert_Element_Subtype'Base'Last in  
  Gaseous_Inert_Element_Subtype
```

### RATIONALE FOR SOLUTION

The expression above employs the membership operator "in" (explained in Exercise 6.3). The attributes 'Base and 'Last are used to refer to the last element of the base type. Gaseous\_Inert\_Element\_Subtype'Base refers to the base type of Gaseous\_Inert\_Element\_Subtype. And Gaseous\_Inert\_Element\_Subtype'Base'Last refers to the last element of that type. The entire expression performs exactly as it reads. It determines whether the last element of the base type of Gaseous\_Inert\_Element\_Subtype is a member of Gaseous\_Inert\_Element\_Subtype. This problem cannot be solved without the use of attributes.

### ALTERNATIVE SOLUTION

```
Gaseous_Inert_Element_Subtype'Last =  
  Gaseous_Inert_Element_Subtype'Base'Last
```

### DISCUSSION

The alternative solution determines whether the last element of Gaseous\_Inert\_Element\_Subtype is the same as the last element of its base type. Gaseous\_Inert\_Element\_Subtype'Last refers to the last element of the subtype, and Gaseous\_Inert\_Element\_Subtype'Base'Last refers to the last element of its base type. If the last element of the base type is in the subtype it must be the last element of the subtype because the subtype cannot extend beyond the range of the base type. This solution is legal and as effective as the first. It is less preferable only because it does not read as well.

This page intentionally left blank

## **Section 8**

# **ARRAY TYPES AND ITERATIVE CONTROL STRUCTURES**

**8.1 Constrained Arrays**

**8.2 Loops**

**8.3 Unconstrained Arrays (String)**

**8.4 Array of Arrays/Array of String**



This page intentionally left blank

## EXERCISE 8.1

### OBJECTIVE

To introduce array types.

### TUTORIAL

An array type is a composite type, which means that an object of an array type is composed of many objects called components. In the case of array types, each component is of the same type. Arrays are used to group similarly typed data into a logical unit. They are useful, for example, for interpreting three-dimensional representations of space.

The syntax for declaring an array type is:

```
type Array_Type_Name is array (Index Subtype Indications)  
  of Component_Subtype;
```

Array\_Type\_Name is the name of the array type and may be any valid identifier. Component\_Subtype is the type of each component of the array, and may be of any type or subtype. The Index Subtype Indications specify the dimensions and index subtype of the array. There is a distinct component for each possible index value. For example, the array type declaration

```
type Path_Connection_Arr_Type is array (Integer range 1 .. 10)  
  of Boolean;
```

establishes a type for an array having ten separate components, each of type Boolean. The Index Subtype Indications specify that the index of Path\_Connection\_Arr\_Type is an integer, having a range from one to ten. Note that the value of the lower and upper bounds of the index range constraint, in this case 1 .. 10, must belong to the specified index subtype (except in the case of the null array, which will be mentioned later in this exercise).

The index subtype must be a discrete type, i.e., an integer or enumeration subtype. The possible values for an index are all the values between the lower and upper bounds, inclusive. The range bounds do not have to be static.

Here are three examples of array type declarations with integer index subtypes:

```
type Density_Arr_Type is array (Integer range -10 .. 25) of Real;  
type Temperature_F_Arr_Type is array (Integer range 32 .. 212)  
  of Integer;  
type Temperature_C_Arr_Type is array (Integer range 0 .. Boiling_Point)  
  of Integer;
```

The first declares an array type called `Density_Arr_Type` having 36 components, each of the declared type `Real`. The index values of `Density_Arr_Type` are in the range -10 to 25. The second declaration defines a type called `Temperature_F_Arr_Type` having 181 components indexed from 32 to 212. `Temperature_C_Arr_Type` has an index range from 0 to `Boiling_Point`, where `Boiling_Point` has a numeric value.

Declared types and subtypes, such as,

```
type Ohms_Type is range 0 .. 5000;  
subtype Velocity_Type is Integer;
```

can be used as index subtypes for array type declarations as follows,

```
type Current_Arr_Type is array (Ohms_Type) of Real;  
type Height_Arr_Type is array (Velocity_Type range 0 .. 100)  
  of Integer;
```

If the array is not specifically unconstrained (unconstrained array types will be discussed later in this exercise) the index has a range constraint of the index subtype.

When the index bounds are both whole numeric and static, the type and the reserved word "range" may be omitted. The index subtype is then implicitly converted to integer. The syntax looks like:

```
type Array_Type_Name is array (Lower .. Upper) of Component_Subtype;
```

where Lower and Upper are the bounds of the index range. For example, the declaration above for `Temperature_F_Arr_Type` could also be written:

```
type Temperature_F_Arr_Type is array (32 .. 212) of Integer;
```

Neither Lower nor Upper can be signed. `Density_Arr_Type` above could not be written in this format because its index lower bound, -10, is not a numeric literal.

The following statements are examples of legal array type declarations:

```
type Line_Arr_Type is array (0 .. 99) of Character;  
type Direction_Arr_Type is array (0 .. 360) of Integer;  
type Deviation_Arr_Type is array (Integer range -1 .. 1) of Real;
```

where Real is a declared type.

The array index may also be an enumeration type. For example, given the type,

```
type Largest_Cities_in_US is (New_York, Los_Angeles, Chicago,  
                               Philadelphia, Detroit,  
                               San_Francisco, Washington, Dallas);
```

an array type for populations may be declared:

```
type Urban_Population_Arr_Type is array (Largest_Cities_in_US)  
of Natural;
```

Urban\_Population\_Arr\_Type is an array having eight components of type Natural. They are indexed by the values of Largest\_Cities\_in\_US. An array type for the population of only the three largest cities could also be created as follows:

```
type Urban_Pop_3_Arr_Type is array  
  (Largest_Cities_in_US range New_York .. Chicago) of Natural;
```

Objects of this array type have three components.

Up to this point, only one-dimensional array types have been discussed. However, an array may have many dimensions. The specification of each dimension is listed in the index subtype specification. For example, consider the following type declaration:

```
type Matrix is array (1 .. 2, 1 .. 10) of Boolean;
```

Matrix is a two-dimensional array type. It is said to have dimension 2 by 10. The first index has an index range from 1 to 2. The second index has a range from 1 to 10. The type of each index is a subtype of Integer. There is a distinct component for every possible sequence of index values. Matrix has 20 components, each of type Boolean.

The dimensions of a multi-dimensional array type can be of different discrete types. For instance, given the following type declarations,

```
type Altitude_Type is range 0 .. 7000;
type Plane_Type is (Cessna, Beachcraft, L1011, P727, P747, DC10,
                    P767, DC9, De_Havilland_Otter);
type Direction_Type is (Roll, Pitch, Yaw);
type Velocity_Type is range 0 .. 800;
type Angle_Type is delta 0.1 range 0.0 .. 360.0;
```

The following multi-dimensional arrays can be declared:

```
type Aircraft_Resistance_Arr_Type is array
  (Integer range 1 .. 250, Altitude_Type, Velocity_Type)
  of Float;
type Aircraft_Attitude_Arr_Type is array
  (Plane_Type, Direction_Type)
  of Angle_Type;
type Sm_Aircraft_Speed_Arr_Type is array
  (Plane_Type range Cessna .. Beechcraft,
   Altitude_Type range 0 .. 2000) of Velocity_Type;
```

When using a declared type or subtype for the index range, it is generally better if the range of that type is the same as the range of the index in the array type definition. It would be preferable, for example, if the index subtype specification of `Sm_Aircraft_Speed_Arr_Type`, above, employed a subtype having the same range as that required for the array type. One could declare the subtypes,

```
subtype Sm_Plane_Type is Plane_Type range Cessna .. Beachcraft;
subtype Low_Altitude_Type is Altitude_Type range 0 .. 2000;
subtype Low_Velocity_Type is Velocity_Type range 0 .. 300;
```

and then declare `Sm_Aircraft_Speed_Arr_Type`, instead, as follows:

```
type Sm_Aircraft_Speed_Arr_Type is array (Sm_Plane_Type,
                                           Low_Altitude_Type)
  of Low_Velocity_Type;
```

This method makes array type declarations easier to read. There are specific plane, altitude, and velocity types for small planes, which may be used elsewhere in the program. There can be no ambiguity in the exact index ranges of `Sm_Aircraft_Speed_Arr_Type`.

An array type may be formed with a null range of index values. A null range is one in which the lower bound exceeds the upper bound. Null arrays have no components. If any dimension of a multi-dimensional array is null the entire array is considered null.

Ada allows the declaration of array types with an unspecified length, called unconstrained arrays. The format of the declaration is:

```
type Array_Type_Name is array (Index_Subtype range < >)
                                of Component_Type;
```

where Index Subtype is the subtype or type of the index, and < > is the compound delimiter called "box". "Range<>" defines an unconstrained range. For example, the declaration,

```
type Message_Switch_Arr_Type is array (Integer range < >) of Boolean;
```

establishes a type Message\_Switch\_Arr\_Type for a one-dimensional unconstrained array of boolean components.

The three-dimensional unconstrained array type,

```
type Air_Space_Arr_Type is array (Integer range< >,
                                   Integer range< >,
                                   Integer range< >)
of Real;
```

might be useful to evaluate the air flow around an aircraft. The dimension corresponds to the x, y, and z axes of a three-dimensional Cartesian graph.

The following declarations could also provide useful unconstrained array types:

```
type Matrix_2_Arr_Type is array (Integer range< >,
                                   Integer range< >)
of Integer;
type Matrix_3_Arr_Type is array (Integer range< >,
                                   Integer range< >,
                                   Integer range< >)
of Integer;
```

Matrix\_2\_Arr\_Type is an unconstrained two-dimensional array of integers, and Matrix\_3\_Arr\_Type is an unconstrained three-dimensional array of integers. If an array is unconstrained, all of its dimensions must be unconstrained.

An unconstrained array type can be used as a base type for the declaration of several array subtypes. For instance, several two-dimensional array subtypes could be declared from the type `Matrix_2_Arr_Type`:

```
subtype Vector_System_2x2_Arr_Type is
    Matrix_2_Arr_Type (1 .. 2, 1 .. 2);
subtype Vector_System_3x3_Arr_Type is
    Matrix_2_Arr_Type (1 .. 3, 1 .. 3);
```

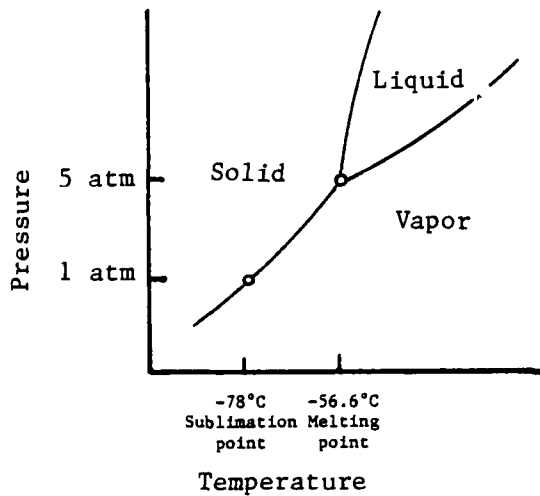
An index constraint may not be applied to an array subtype that already has an index constraint. One could not write:

```
subtype Vector_System_2x3_Arr_Type is
    Vector_System_3x3_Arr_Type (1 .. 2, 1 .. 3);
```

because `Vector_System_3x3_Arr_Type` already has an index range constraint. Remember finally, that the array type declarations described so far are type declarations. They establish a form for an array, but do not create the array itself. Object declarations, to be discussed in the next exercise, provide working array objects.

### PROBLEM

Below is a phase diagram for carbon dioxide with pressure and temperature on the y and x axes. Write all type declarations needed to define an array type representing an estimation of this graph. It should store the state (solid, liquid, or vapor) of the carbon dioxide for any given temperature and pressure (use integer values; 1 atm  $\leq$  pressure  $\leq$  10 atm and  $-78\text{ C} \leq$  temperature  $\leq 0\text{ C}$ ).



The phase diagram for carbon dioxide  
(not drawn to scale)



This page intentionally left blank

## SOLUTION TO EXERCISE 8.1

### SOLUTION

```
type State_Type is (Solid, Liquid, Vapor);
type Pressure_Type is range 1 .. 10;
type Temperature_Type is range -78 .. 0;
type State_Diagram_Type is array (Pressure_Type, Temperature_Type)
  of State_Type;
```

### RATIONALE FOR SOLUTION

State\_Type is declared an enumeration type for the three possible states, solid, liquid, and vapor. State\_Diagram\_Type is declared an array type with components of State\_Type. State\_Diagram\_Type is two-dimensional, simulating a graph, with two indices of declared integer types Pressure\_Type and Temperature\_Type. The bounds of each index correspond to the range of possible values for pressure and temperature.

### ALTERNATIVE SOLUTION

```
type State_Type is (Solid, Liquid, Vapor);
type Pressure_Type is range 1 .. 10;
type Temperature_Type is range -78 .. 0;
type State_Diagram_Type is array (Pressure_Type,
                                   Temperature_Type,
                                   State_Type)
  of Boolean;
```

### DISCUSSION

The alternate solution declares a three-dimensional array type State\_Diagram\_Type with an index for each Pressure\_Type, Temperature\_Type, and State\_Type. The components of the array are of type Boolean and indicate whether, for a given temperature, pressure, and state, that state is appropriate. In other words, for each possible pair of temperature and pressure values, there are three state components, two of which will be marked false, and one true.

The alternate solution is legal and accurate, but it is unnecessary to create an array with three dimensions. A two-dimensional array depicts the graph most effectively, giving a state type at each point on the graph.

This page intentionally left blank

## EXERCISE 8.2

### OBJECTIVE

To illustrate array objects.

### TUTORIAL

Object declarations for the array types explained in the previous chapter have the format:

Array\_Name : Array\_Type\_Name [(Index Constraint)];

Array\_Name is the identifier for the array being declared, and Array\_Type\_Name is its type. The Index\_Constraint is included only if Array\_Type\_Name is unconstrained. It must be the constraint of the Index\_Subtype\_Indications found in the Array\_Type\_Name definition.

One could create objects of the array types from the last exercise,

```
type Vector_System_Arr_Type is array
  (Integer range<>, Integer range<>) of Integer;
subtype Vector_System_2x2_Arr_Type is
  Vector_System_Arr_Type (Integer range 1 .. 2, Integer range 1 .. 2);
subtype Vector_System_3x3_Arr_Type is
  Vector_System_Arr_Type (Integer range 1 .. 3, Integer range 1 .. 3);
```

as follows:

```
Vector_System_2x2 : Vector_System_2x2_Arr_Type;
Vector_System_3x3 : Vector_System_3x3_Arr_Type;
```

These declarations have created two array objects, one called Vector\_System\_2x2 and the other Vector\_System\_3x3. The first is of type Vector\_System\_2x2\_Arr\_Type, and the second is of type Vector\_System\_3x3\_Arr\_Type.

The declaration,

```
Vector_System_5x5 : Vector_System_Arr_Type (1 .. 5, 1 .. 5);
```

creates an array object from the unconstrained type Vector\_System\_Arr\_Type. The index range constraint is given in the object declaration.

One could also declare an array object using the following syntax:

Array\_Name : array (Index Constraint) of Component\_Subtype;

The declaration,

```
Vector_System_2x3_A : array (Integer range 1 .. 2,  
                             Integer range 1 .. 3) of Integer;
```

creates an array that is of an anonymous type. It is similar to the array created by the following two declarations:

```
type Vector_System_2x3_Arr_Type is array (1 .. 2, 1 .. 3) of Integer;  
Vector_System_2x3_B : Vector_System_2x3_Arr_Type;
```

Vector\_System\_2x3\_A and Vector\_System\_2x3\_B have the same indices and component type, but they are not of the same type. Vector\_System\_2x3\_B is of a named type, Vector\_System\_2x3\_Arr\_Type, while Vector\_System\_2x3\_A is of an anonymous array type. Therefore, Vector\_System\_2x3\_A and Vector\_System\_2x3\_B could not be assigned to one another.

The following declaration,

```
Vector_System_4x4_A,  
Vector_System_4x4_B : array (1 .. 4, 1 .. 4) of Integer;
```

creates two arrays of an anonymous array type. They have the same specifications and are declared in the same statement, but they still belong to two distinct types. Vector\_System\_4x4\_A could not be assigned to Vector\_System\_4x4\_B, and vice versa.

But by declaring,

```
Vector_System_4x4_Arr_Type is array (1 .. 4, 1 .. 4) of Integer;  
Vector_System_4x4_A, Vector_System_4x4_B : Vector_System_4x4_Arr_Type;
```

Vector\_System\_4x4\_A and Vector\_System\_4x4\_B are of the same type, and may be assigned to one another or compared under array operators. The use of a type name enables the assignment of whole arrays that have been separately declared. It is the only way to create many array objects of the same type. Also, an anonymous array declaration may not be unconstrained.

Like all object declarations, array object declarations may include an expression of initial values. When an initial expression is given for an array object there must be an initial value for every component. The format is:

```

Array_Name : Array_Type_Name (Index Constraint) := Initial_Array_Value;

or

Array_Name : array (Index Constraint) of Component_Type
                                     := Initial_Array_Value;

```

Initial\_Array\_Value may be an array aggregate, a string literal, or a previously declared array. (Strings will be discussed in Exercise 8.4.) An array aggregate is a composite of the values of each component. It can be written in named or positional notation. An example of initialization to an aggregate in positional notation is:

```

type Boolean_Arr_Type is array (1 .. 8) of Boolean;
Boolean_Array : Boolean_Arr_Type :=
    (True, False, True, False, False, False, True, False);

```

An aggregate for a two-dimensional array

```

type Boolean_Matrix_Type is array (1 .. 3, 1 .. 8) of Boolean;
Boolean_Matrix : Boolean_Matrix_Type;

```

can be written in positional notation as

```

((True, True, False, True, False, False, False, False)
 (True, True, False, True, False, False, False, False)
 (True, False, True, True, True, True, True, True))

```

Note that an aggregate for an n-dimensional array is made up of several one-dimensional aggregates, or subaggregates.

In an aggregate using named notation each index value is explicitly given. The component value is preceded by the corresponding index value and =>. The components may be listed in any order. The rules are similar to those for choices in case statements, allowing .. (range), | (or), and "others." Each component may be listed only once in the aggregate.

The above aggregate for Boolean\_Matrix written in named notation would be

```

(1 .. 2 => (1 | 2 | 4 => True, 3 => False, 5 .. 8 => False),
 3      => (1 => True, 2 => False, 3 .. 8 => True))

```

The reserved word "others" may be used in either named or positional notation. It refers to "all the rest" and must appear as the last item in the aggregate. The above aggregate could also be written:

```
(1 .. 2 => (1 | 2 | 4 => True, 3 => False, 5 .. 8 => False),
 3      => (2 => False, others => True))
```

When "others" is used, the exact bounds of the index must be clear from the context.

All ranges and values before the => must be static, except when there is only one alternative with a single choice. For example,

```
A : array (1 .. Var) of Float := (1 .. Var => 0.0);
```

where Var is a variable.

An aggregate with only one component must use named notation. This rule is meant to prevent unnecessary ambiguity. It is difficult to know whether

(5)

is an aggregate of a single integer, or a numeric literal surrounded by parentheses. Aggregates having more than one component must use either all positional or all named notation. ("Others" may be used with positional notation.)

The following aggregates are equivalent:

```
(1 => (1.0, 0.0),
 2 => (0.0, 1.0))

((1 => 1.0, 2 => 0.0),
 (1 => 0.0, 2 => 1.0))

(1 => (1 => 1.0, 2 => 0.0),
 2 => (0.0, 1.0))
```

Constant arrays may be formed by using the format:

```
Array_Name : constant Array_Type_Name (Index_Constraint)
              := Initial_Value;
```

or

```
Array_Name : constant array (Index_Constraint)
              of Component_Subtype := Initial_Value;
```

"Constant" is a reserved word. The initial value must be included in a constant declaration.

Constant arrays are most often used for look-up tables. Here are some examples:

```
Signal_Frequency_Arr : constant array (Integer range 1 .. 7)
                        := (3360, 4980, 5510, 5690, 5770, 5920, 7100);

type Model_Type is (CT301, CT800, RT222);
Mass_Arr : constant array (Model_Type,
                           Integer range 1 .. 3) of Float
:= (CT301 => (23.0, 45.3, 89.1),
    CT800 => (25.4, 50.0, 77.6),
    RT222 => (28.9, 30.2, 65.0));
```

The values of the components of Signal\_Frequency\_Arr and Mass\_Arr now cannot be changed.

An initialized array declaration need not always include the range bounds of the index. The bounds may be implicitly determined from the initial value. For example, the declarations

```
type Matrix_Arr_Type is array
    (Natural range <>,
     Natural range <>)
  of Integer;
Matrix_3x3 : constant Matrix_Arr_Type := (1 => (3, 4, 5),
                                           2 => (4, 8, 9),
                                           3 => (2, 9, 0));
```

create a constant array Matrix\_3x3. The index bounds of Matrix\_3x3 are deduced from the initial aggregate. Note that the index subtype is Natural, not Integer. If the initial value were not written in named notation, and Integer were used for the index subtype, the compiler would assign to the lower index bound the value of Integer'First. It is generally a better idea to use named notation for the initial aggregate.

Aggregates are not the only means of assignment to arrays. A value may be assigned to individual array components or groups of components. To access a component of an array object for assignment or evaluation write:

```
Array_Name ( First_Index, ... , Nth_Index )
```



For example, the third element in the first row of Boolean\_Matrix could be accessed by writing

```
Boolean_Matrix(1,3)
```

and it could be assigned the value True as follows:

```
Boolean_Matrix(1,3) := True;
```

One may also refer to a portion of a one-dimensional array, called a slice. For example a slice of Boolean\_Array (declared earlier),

```
Boolean_Array (3 .. 7)
```

accesses the third through seventh components of that array. The assignment,

```
Boolean_Array (3 .. 7) := (True, True, True, True, True);
```

assigns an aggregate to that portion of the array. A slice may only be applied to a one-dimensional array.

The basic operations for objects of array types include indexed components and membership tests. Additionally there are four attributes: 'First, 'Last, 'Range, and 'Length. Each of these attributes may or may not include a static argument specifying a dimension of the array.

Below is a list of the operations of these attributes.

'First 'First(N)	yields the lower bound of the index range.
'Last 'Last(N)	yields the upper bound of the index range.
'Range 'Range(N)	yields the index range.
'Length 'Length(N)	yields the number of values of the index range.

So, for the array object Boolean\_Array these attributes would return the following values.

```
Boolean_Array'First = 1
```

```
Boolean_Array'Last = 8
```

```
Boolean_Array'Range = 1 .. 8
```

```
Boolean_Array'Length = 8
```

For the multi-dimensional array object Boolean\_Matrix, the attributes with arguments would return the following:

```
Boolean_Matrix(2)'First = 1, the lower bound of the second index
Boolean_Matrix(1)'Last  = 3, the upper bound of the first index
Boolean_Matrix(2)'Range = 1 .. 8, the range of the second index
Boolean_Matrix(1)'Length = 3, the length of the first index
```

If the index is not specified, it is assumed to be the first. For example,

```
Boolean_Matrix'Range = 1 .. 3, the range of the first index.
```

'Range can be used in an aggregate declaration:

```
A : array (1 .. 100) of Float := (1 .. 100 => 0.0);
```

could also be written

```
A : array (1 .. 100) of Float := (A'Range => 0.0);
```

In summary, the formats for declaring array objects explained here are:

#### Named Types

```
Array_Name : Array_Type_Name;
```

```
Array_Name : Array_Type_Name (Index Constraint); -- used only if
                                                    -- Array_Type_Name
                                                    -- is unconstrained
```

```
Array_Name : constant Array_Type_Name := Initial_Array_Value;
```

```
Array_Name : constant Array_Type_Name (Index Constraint)
                := Initial_Array_Value; -- used only
                -- if Array_Type_Name
                -- is unconstrained
```

#### Anonymous Types

```
Array_Name : array (Index Constraint) of Component_Subtype;
```

```
Array_Name : constant array (Index Constraint) of Component_Subtype
                := Initial_Array_Value;
```

This page intentionally left blank

### PROBLEM

Write a function to compute the determinant of a 3 x 3 matrix of real numbers.  
Also write accompanying type declarations.

(For a matrix,

$$\begin{pmatrix} A1 & B1 & C1 \\ A2 & B2 & C2 \\ A3 & B3 & C3 \end{pmatrix}$$

the determinant = (A1 \* B2 \* C3) + (A2 \* B3 \* C1) + (A3 \* B1 \* C2)  
- (A3 \* B2 \* C1) - (A2 \* B1 \* C3) - (A1 \* B3 \* C2).)

This page intentionally left blank

## SOLUTION TO EXERCISE 8.2

### SOLUTION

```
type Real is digits 4;
type Matrix_Type is array
  (Integer range 1 .. 3,
   Integer range 1 .. 3)
  of Real;
function Determinant (M : Matrix_Type) return Real is
  D : Real;
begin -- Determinant
  D := (M(1, 1) * M(2, 2) * M(3, 3))
    + (M(1, 2) * M(2, 3) * M(3, 1))
    + (M(1, 3) * M(2, 1) * M(3, 2))
    - (M(1, 3) * M(2, 2) * M(3, 1))
    - (M(1, 2) * M(2, 1) * M(3, 3))
    - (M(1, 1) * M(2, 3) * M(3, 2));
  return D;
end Determinant;
```

### RATIONALE FOR SOLUTION

The matrix is represented by the two-dimensional array type `Matrix_Type`. `Matrix_Type` is of dimension  $3 \times 3$ , corresponding to a  $3 \times 3$  matrix. The components are of the declared floating point type `Real`. The function is called `Determinant`. It accepts one parameter of type `Matrix_type` called `M`, and returns the determinant as a `Real` type. Within the function, `D` is declared an object of type `Real` to store the value of the determinant, and it is assigned the value of the formula for `M`. The formula for the determinant is written using the array notation for referencing a component in a two-dimensional array, which is quite readable and corresponds to the actual formula well.

Notice that we could not have used an anonymous array type to represent the matrix, because a named type was required for the specification of the formal parameters of the function `Determinant`.

Using an unconstrained array type for the formal parameter of `Determinant`, we could have written a function that would calculate the determinant for a matrix of any dimension. `Matrix_Type` would be:

```
type Matrix_Type is array (Integer range<>, Integer range<>) of Real;
```

and the specification of `Determinant` would be:

```
function Determinant (M : Matrix_Type) return Real
```

The function body would require some loops, which are covered in the next exercise.

Without an array type, each element of the matrix would have to be separately declared, and the function Determinant would require nine formal parameters, which would be even more ridiculous for a 4 x 4 or 5 x 5 matrix.

#### ALTERNATIVE SOLUTION

```
type Real is digits 4;
type Vector_Type is array (Integer range 1 .. 3) of Real;
V1 : Vector_Type;
V2 : Vector_Type;
V3 : Vector_Type;

function Determinant (V1, V2, V3 : Vector_Type) return Real is
    D : Real;
begin -- Determinant
    D := (V1(1) * V2(2) * V3(3))
        + (V1(2) * V2(3) * V3(1))
        + (V1(3) * V2(1) * V3(2))
        - (V1(3) * V2(2) * V3(1))
        - (V1(2) * V2(1) * V3(3))
        - (V1(1) * V2(3) * V3(2));
    return D;
end Determinant;
```

#### DISCUSSION

Instead of a two-dimensional array the alternate solution uses three one-dimensional arrays, each with three components. These arrays can be thought to represent the three vectors forming the matrix.

This solution is very similar to the first and is acceptable. The notation for the formula is quite readable. However, the dimensions of the matrix in the first solution can more easily be changed. One might as well use a two-dimensional array since the language allows it. It is a more convenient and less error-prone way to represent matrices.

## EXERCISE 8.3

### OBJECTIVE

To demonstrate loops.

### TUTORIAL

Loop constructs are used to execute a particular sequence of statements zero or more times. A simple loop might look like this:

```
loop
  Get (Current_Character);
  exit when Current_Character = ' ';
end loop;
```

This construct will execute the first statement, "Get (Current\_Character);" and then it will check the condition in the exit statement. If it is True, i.e. Current\_Character = ' ', then control exits the loop and passes to the next statement. If it is not True, control passes to the top of the loop, and "Get (Current\_Character);" is again executed. The statement is executed iteratively until Current\_Character = ' '.

A loop statement usually has an iteration scheme, which is specified before the sequence of statements. It determines how many times the following statements will be executed. There are two kinds of statements containing iteration schemes: "while" and "for".

A while loop has the format:

```
while Condition
loop
  Sequence of Statements;
end loop;
```

Condition is an expression that evaluates to a Boolean value. Sequence of Statements is any number of statements to be executed. The loop above for reading in characters until a ' ' is found could also be written:

```
Get (Current_Character);
while Current_Character /= ' ' loop
  Get (Current_Character);
end loop;
```



In this statement the condition is evaluated before each execution of the succeeding statements. Control passes out of the loop as soon as `Current_Character /= ' '` is False. There is a small difference between this construct and the first loop. The while statement will initially evaluate the condition before the execution of the get statement in the loop, but the other loop statement will check the condition only after execution of the get statement. This difference depends upon the value of `Current_Character` before entering the loop statement.

Here is another example of a while loop:

```
while not Bin_Full (Bin)
loop
    Add_Element (Bin);
    Verify_Elements (Bin);
end loop;
```

`Bin_Full` is a function returning a Boolean value, True if Bin is full. `Add_Element` adds an element to Bin, and `Verify_Elements` verifies the elements in Bin and appropriately changes the contents. These two procedures are invoked until Bin is full. The use of a while loop allows a condition based on the bin size to control the execution of two procedures that change the bin size.

In the above example, it is not possible to know how many times the procedures in the loop will be invoked before the bin becomes full. The while loop executes while a certain condition is True. A for loop, on the other hand, is used when we know how many times we want to execute the statements within the loop. It has the format:

```
for Loop_Parameter in Discrete_Range
loop
    Sequence of Statements;
end loop;
```

Loop\_Parameter is incremented by one within Discrete\_Range, and Sequence of Statements is executed for each value of the loop parameter. Both the loop parameter and the range must be of discrete types.

A for loop might be used, for instance, to calculate the value of an infinite series to a specified degree of accuracy. For example, to calculate  $e$  using the series formula  $e = 1/0! + 1/1! + 1/2! + 1/3! + \dots + 1/n!$  (assume  $0! = 1$ ) to a fairly high degree of accuracy we could write a loop to add the first 50 terms of this series:

```

procedure Compute_e is
    type Real is digits 12 range 0.0 .. 3.0;
    e : Real := 1.0;
    Term : Real := 1.0;
    Accuracy : Integer := 50;

begin -- Compute_e

    for I in 1 .. Accuracy
    loop
        Term := Term / Real(I);
        e := e + Term;
    end loop;

end Compute_e;

```

The loop parameter I is declared implicitly. It cannot be declared in a variable declaration. The scope of I extends from its specification to the end of the loop, and it is only visible within the loop, i.e. it cannot be referenced outside of the loop. Its value may only be saved by assigning it to a variable within the loop.

The value of the loop parameter cannot be altered in the sequence of statements within the loop. A statement like,

```
I := I + 1;
```

would not be allowed inside the loop. And the loop parameter may not be given as an "out" or "in out" parameter of a procedure.

The type of the loop parameter is determined by the range given. The type of the bounds of the range must therefore be unambiguous. If the range is null the sequence of statements is executed zero times.

The loop parameter is implicitly incremented by one. It cannot step by any other number. But, the parameter may also be decremented by one as follows:

```

for J in reverse 1 .. 10
loop
    Last_Space (K_Buffer (J)) := Next_Value;
end loop;

```

The reserved word "reverse" specifies that J is decremented by one, having an initial value of 10, and a final value of 1. Note that the range is always written in ascending order. The following iterative scheme,

```
for K in reverse 10 .. 1
loop
```

defines a null range, exactly as does

```
for K in 10 .. 1
loop
```

The statements inside a null loop are never executed.

A loop statement may have a loop identifier as in the following function:

```
type Arr_Type is array (Integer range<>) of Integer;

function Integer_Sum (N : Arr_Type) return Integer is
    Sum : Integer := 0;
begin -- Integer_Sum
    Sum_Loop:
        for I in N'Range
        loop
            Sum := Sum + N (I);
        end Sum_Loop;
    return Sum;
end Integer_Sum;
```

The loop identifier is Sum\_Loop. Note that if a name is given at the beginning of the loop, it must also be given at the end. Loop identifiers are used to improve readability, perhaps to alleviate confusion between nested or closely grouped loops. They also specify the loop to be exited when an exit statement is used.

Either for or while statements should be adequate for most looping situation, but there are two other statements that may be used: "goto" and "exit." An exit statement terminates the current loop and directs control to the point after the next appropriate "end loop". For example, a search of a three-dimensional array,

```

procedure Search_3_d_Array is
    type Arr_Type is array (Integer range<>,
                             Integer range<>,
                             Integer range<>) of Character;
    Arr : Arr_Type (1 .. 10, 1 .. 10, 1 .. 10);
    First_Index, Second_Index, Third_Index : Integer;
begin -- Search_3_d_Array
    .
    -- Statements to assign values to Arr.
    .
    Search:
        for I in Arr'Range
        loop
            for J in Arr'Range(2)
            loop
                for K in Arr'Range(3)
                loop
                    if Arr(I, J, K) = '*' then
                        First_Index := I;
                        Second_Index := J;
                        Third_Index := K;

                        exit Search;

                    end if;
                end loop;
            end loop;
        end loop Search;
    end Search_3_d-Array;

```

can use an exit statement to exit three nested loops when the first '\*' is found. Note that the exit statement includes a loop identifier. Control, therefore, exits the loop with that identifier. The loop identifier is used in the above example to exit three nested loops. If the loop name Search were not included in the exit statement, control would continue at the top of the second for statement.

In the above example, three integers are declared, First\_Index, Second\_Index, and Third\_Index, to store the values of the array indices when the '\*' is found.

They must be declared because the loop parameters are only visible within the loop, and unless their values are assigned to a declared variable, the location of the '\*' would not be known at the end of the search.

A "when" may also be used in an exit statement to state a condition for exit. For example,

```
    loop
      Inhale;
      Exhale;

      exit when Dead;

    end loop;
```

It is preferable to use the conditional exit over the unconditional exit.

A goto statement directs control to a target statement named by a label. The above linear search could be written with a goto as follows:

```
    for I in Arr'Range
    loop
      for J in Arr(2)'Range
      loop
        for K in Arr(3)'Range
        loop
          if Arr(I, J, K) = '*' then
            First_Index := I;
            Second_Index := J;
            Third_Index := K;
            goto Found_Continue;
          end if;
        end loop;
      end loop;
    end loop;
    Put ("No * found");
    << Found_Continue >> null;
```

The label is << Found\_Continue >>. The goto directs control out of the loop to the statement labeled by << Found\_Continue >>. A label should not be confused with a loop identifier. A label is only the destination of a goto statement.

It is definitely not recommended programming practice to use a goto statement. It is mentioned here only because it is part of the language.

### PROBLEM

Write two functions, one that determines whether a matrix is lower triangular, and another that calculates the determinant of a triangular matrix. In a lower triangular matrix, all of the elements above the main diagonal are zero:

$$\begin{pmatrix} a_{11} & 0 & 0 & 0 & \dots & 0 \\ & a_{22} & 0 & 0 & \dots & 0 \\ & & a_{33} & 0 & \dots & 0 \\ & & & \ddots & & \\ & & & & \ddots & \\ & & & & & \dots & a_{nn} \end{pmatrix}$$

and the determinant is the product of the elements on the main diagonal,  $a_{11} * a_{22} * a_{33} * \dots * a_{nn}$ . Each function should accept a matrix of any size (you may assume that it is square). The index ranges must start with 1.

This page intentionally left blank

## SOLUTION TO EXERCISE 8.3

### SOLUTION

```
type Real is digits 5;
type Matrix_Type is array (Integer range<>, Integer range<>) of Real;

function Triangular (M : Matrix_Type) return Boolean is
    K : Integer := M'First;           -- lower bound of row index
begin -- Triangular
    for Col in M'First +1 .. M'Last    -- columns left to right
    loop
        for Row in M'First .. K       -- rows top to K
        loop
            if M(Row,Col) /= 0.0 then
                return False;
            end if;
        end loop;
        K := K + 1;
    end loop;
    return True;
end Triangular;

function Determinant_Of_Triangular (M : Matrix_Type) return Real is
    D : Real := 1.0;
begin -- Determinant_Of_Triangular
    for I in M'Range
    loop
        D := D * M(I, I);
    end loop;
    return D;
end Determinant_Of_Triangular;
```

### RATIONALE FOR SOLUTION

The first function Triangular accepts a parameter M of unconstrained array type Matrix\_Type and returns a boolean value indicating whether M is a triangular



matrix. A variable K is declared within the function. It is the lower bound of the row index. The function moves through the matrix from left to right going down the columns. Because only the elements above the main diagonal need be checked, the first column is not viewed at all, only the top element in the second column is, the top two in the third column, etc. K successively increments by one following the elements just above the main diagonal in each column. It is used as the upper bound for the loop parameter that searches each row.

The first and outside loop parameter Col moves from the second column across the matrix to the last column. For every value of Col the inside loop parameter Row searches the column from the top row to K for a non-zero element. If one is found the matrix cannot be triangular and False is returned. After the column is searched K is incremented and control returns to the top of the outside loop, moving the search to the next column. (Note that K is not a loop parameter, and may, therefore, be assigned a value within the loop body.) If the entire matrix has been searched without a non-zero element being found, True is returned.

The second function Determinant\_of\_Triangular also accepts a parameter M of unconstrained array type Matrix\_Type. It returns the real value of the determinant of M. D is declared of type Real to store the resulting determinant, and is initialized to 1.0. It is assigned the value of the product of the elements on the main diagonal (all \* a22 \* a33 \* ... \* aNN) using a for loop. The loop parameter I is incremented from 1 to the last element of the first index of M. As I is incremented D is multiplied by each new matrix element. When every element on the diagonal has been multiplied, control leaves the loop, and the value of D is returned.

Note that the attribute 'Range is used to express the entire range of a loop parameter. In the first function M'Range was not used for Col because Col starts at one past M'First so the attribute 'Last was used to refer to the upper bound of the loop parameter range. Without attributes these functions would require a second "in" parameter that would indicate the length of the matrix M.

Both of these functions use a for loop to move an index through the matrix. This could also have been accomplished using a loop parameter that moved from the last column to the first and the bottom row to the top using a reverse loop statement. For example the iteration schemes for the outside loop in Triangular could also be written:

```
for Col in reverse M'First +1 .. M'Last
loop
  for Row in reverse 1 .. K
  loop
```

Of course K would have to be initialized to M'Last - 1 and be decremented inside the loop. A reverse search would work equally as well, but there is, in this case, no compelling reason for it to be done that way.

## ALTERNATIVE SOLUTION

```
type Real is digits 5;
type Matrix_Type is array (Integer range<>, Integer range<>) of Real;
function Triangular (M : Matrix_Type) return Boolean is
    Col : Integer := M'First + 1;      -- column index, starts at second
    Row : Integer := M'First;          -- row index, starts at top
    K : Integer := M'First;            -- upper bound of row index
begin -- Triangular
    while Col <= M'Last                 -- second to last column
    loop
        while Row <= K                 -- first to Kth row
        loop
            if M(Row, Col) /= 0 then
                return False;
            end if;
            Row := Row + 1;
        end loop;
        K := K + 1;
        Row := M'First;                -- reset Row
        Col := Col + 1;
    end loop;
    return True;
end Triangular;

function Determinant_Of_Triangular (M : Matrix_Type) return Real is
    D : Real := 1.0;
    I : Integer := M'First;            -- index for matrix
begin -- Determinant_Of_Triangular
    while I <= M'Last                  -- first to last column
    loop
        D := D * M(I, I);
        I := I + 1;
    end loop;
    return D;
end Determinant_of_Triangular;
```

## DISCUSSION

In the alternate solution while loops are used to move an index through the matrix. The equivalent of the loop parameters in the first solution are here declared separately, and incremented until they exceed the range of the matrix. This solution is difficult to read, and completely inappropriate. Since the range of the matrix M can be referenced using attributes 'Last or 'Range the number of iterations of the loop statements is known, and a for loop should be used. A while loop is most appropriate when the number of iterations depends upon a condition that changes unpredictably.

This page intentionally left blank

## EXERCISE 8.4

### OBJECTIVE

To demonstrate strings.

### TUTORIAL

Objects of predefined type String are one-dimensional arrays with components of predefined type Character, and an index of the predefined subtype Positive.

subtype Positive is Integer range 1 .. Integer'Last;  
type String is array (Positive range<>) of Character;

One can create an object of type String using the format:

```
String_Name : String (Lower_Bound .. Upper_Bound);
```

String\_Name is the identifier for the String object being declared.  
Lower\_Bound and Upper\_Bound are the bounds defining the length of String\_Name. Both must be of type positive. A string may be initialized as follows:

```
String_Name : String (Lower_Bound .. Upper_Bound) := String_Literal;
```

where String\_Literal is a string literal. A string literal can be written as a sequence of characters enclosed in quotes, such as

```
"ANSI/MIL-STD-1815 A"
```

Since a string is an array of characters, we could also write the above literal as an array aggregate:

```
('A','N','S','I','/','M','I','L','-','S','T','D','-','1','8','1','5',  
 ' ','A')
```

or equivalently, and in a less readable fashion:

```
(1 | 19 => 'A', 2 => 'N', 3 | 10 => 'S', 4 | 7 => 'I',  
 5 => '/', 6 => 'M', 8 => 'L', 9 | 13 => '-', 11 => 'T',  
 12 => 'D', 14 | 16 => '1', 15 => '8', 17 => '5', 18 => ' ')
```

The first method with quotes is easier and much more readable.

To put a quote mark inside a string write double quote marks. For example,

```
Famous_Quote : String (1 .. 26) := """"Dooby Dooby Doo"" -Sinatra";
```

looks like

```
"Dooby Dooby Doo" -Sinatra
```

In an initialized string declaration the initial string literal must be the same length as the declared object. One could declare a string object for the reference manual as follows:

```
Reference_Manual : String (1 .. 19) := "ANSI/MIL-STD-1815 A";
```

Reference\_Manual is declared an object of type String with 19 characters, initialized to "ANSI/MIL-STD-1815 A". One could then assign another string literal to Reference\_Manual:

```
Reference_Manual := "Copyright U.S. Govt";
```

Any string literal that is assigned to Reference\_Manual must be 19 characters long.

The length of a constant string may be determined by the initial value. For instance,

```
Telephone_Book : constant String := "Boston Area Yellow Pages";
```

declares a string called Telephone\_Book which has 24 characters.

Slices may be assigned to one another even though they have different indices, as long as their lengths are the same. For example,

```
Reference_Manual (16 .. 19) := Telephone_Book (8 .. 11);
```

making Reference\_Manual "Copyright U.S. Area."

In addition to assignment, there are three other string operations: equality/inequality, lexical ordering, and catenation.

Equality/inequality operators, "=" and "/=", can be used to compare strings. For example, given the strings,

```
Book_1 : String (1 .. 13) := "War and Peace";  
Book_2 : String (1 .. 20) := "Crime and Punishment";  
Book_3 : String (1 .. 16) := "Fathers and Sons";
```

the expression,

```
Book_1 /= Book_2
```

is True, and

```
Book_2 (7 .. 9) = Book_3 (9 .. 11)
```

is also True. Note that strings of differing lengths may be compared under equality/inequality operators, but of course, they are never equal.

The catenation operator, &, allows the programmer to catenate several string objects and literals. For example, given the strings,

```
Singer_1 : String (1 .. 5) := "Peter";  
Singer_2 : String (1 .. 4) := "Paul";  
Singer_3 : String (1 .. 4) := "Mary";
```

we could create one string,

```
Group : String (1 .. 21) := Singer_1 & ", " & Singer_2 &  
                                ", and " & Singer_3;
```

making Group "Peter, Paul, and Mary". We could also declare a string,

```
DoA : String (1 .. 25) := "Department " & "of " & "Agriculture";
```

Note that spaces separating the words must be included in the strings.

The lexical ordering operators are < , > , <=, and >=. They determine the order of words alphabetically. However, upper case letters precede all lower case letters. For example, the following expressions are all True:

```
"Zeta" < "alpha"  
"Alpha" < "Beta"  
"Alpha Romeo" > "Alpha"  
"Beta" >= "Beta"  
"Gamma" <= "Gamma Rays"  
"10" < "2"
```

Note that in lexical ordering, strings may not be compared to characters.

The following comparison,

```
"Alpha" > 'A'
```

is not allowed.

String attributes are the same as those for all array types:

```
'First  
'Last  
'Range  
'Length
```

Below are some examples of attribute operations:

```
Secretary_Of_Treasury : String (1 .. 7) := "D.Regan";
```

```
Secretary_Of_Treasury'First      -- = 1  
Secretary_Of_Treasury'Last      -- = 7  
Secretary_Of_Treasury'Range     -- = 1 .. 7  
Secretary_Of_Treasury'Length    -- = 7
```

I/O for strings is handled by the procedures,

```
procedure Put (Item : in String);  
procedure Get (Item : out String);  
procedure Put_Line (Item : in String);  
procedure Get_Line (Item : out String; Last : out Natural);  
procedure New_Line (Spacing : in Positive_Count := 1);
```

which are provided by the predefined package Text\_IO. Last gives the length of the string read in. In the following example,

```
with Text_IO; use Text_IO;           -- makes Text_IO procedures available
procedure Weekly_Status is
```

```
    Name_Of_Program : String (1 .. 100);
    Index : Natural;
    Status : Character;
```

```
begin -- Weekly_Status
```

```
    Put_Line ("Type in name of program");
    Get_Line (Name_Of_Program, Index);
    New_Line;
    for I in 1 .. 3 loop
        Put ("Status at beginning of week (A, B, C, or D): ");
        Get (Status);
        New_Line;
    end loop;
```

```
end Weekly_Status;
```

Name\_Of\_Program is declared to be 100 characters long, an arbitrary safe length. If the length of the name that the user types in has only 9 characters, Index is used to refer only to the appropriate slice of Name\_Of\_Program. Put\_Line prints on the terminal the succeeding string followed by a carriage return. Get\_Line reads in up to 100 characters until a carriage return is encountered. The character string is stored in Name\_Of\_Program, and the length of the string is in Index. New\_Line prints a carriage return on the terminal or skips a line on a printer. It may take an argument that determines how many carriage returns are printed. If there is no argument one is assumed. Put prints the succeeding string onto the terminal without a carriage return. The Get reads in one character and stores it in Status. Get ignores carriage returns. The Get, Put, and New\_Line commands at the end are repeated three times, presumably for three weeks of information.

The printout for the above sequence of code would look like this (text typed in by the user is underlined):

Type in name of program

Urban Renewal Plan 3

Status at beginning of week (A, B, C, or D): C

Status at beginning of week (A, B, C, or D): B

Status at beginning of week (A, B, C, or D): A



This page intentionally left blank

### PROBLEM

Write a program that reads in from the terminal a square matrix of any size and its dimension, and if it is triangular, prints its determinant. Assume that the functions written for the problem in Exercise 8.3 are in the package below. If it is not triangular the program should print a message saying so. For real and integer I/O use the functions and procedures in the package:

```
package Matrix_Operations is

  type Real is digits 5;
  type Matrix_Type is array (Integer range <>,
                             Integer range <>)
    of Real;

  function Real_From_Terminal return Real;           -- returns a real
                                                    -- number read
                                                    -- from the terminal

  function Integer_From_Terminal return Integer;      -- returns an
                                                    -- integer read
                                                    -- from the terminal

  procedure Print_Real (R : in Real);                -- prints a real
                                                    -- number on the
                                                    -- screen

  procedure Print_Integer (I : in Integer);          -- prints an integer
                                                    -- on the screen

  function Triangular (M : Matrix_Type) return Boolean;

  function Determinant_of_Triangular (M : Matrix_Type) return Real;

end Matrix_Operations;
```

The purpose of this exercise is to utilize strings most effectively in interaction with the user.

This page intentionally left blank

## SOLUTION TO EXERCISE 8.4

### SOLUTION

```

with Text_IO;          use Text_IO;
with Matrix_Operations; use Matrix_Operations;
procedure Calculate_Determinant_Of_Triangular_Matrix is

    Dimension    : Integer;          -- dimension of matrix

    procedure Get_Data_and_Do_Calc (Dimension : in Integer) is
        Matrix : Matrix_Type (1 .. Dimension, 1 .. Dimension);
        Determinant : Real;

    begin

        Put_Line ("Enter elements going down columns, left to right " &
            "(elements must be real):");
        for Col in Matrix'Range
            loop
                -- fill Matrix
                for Row in Matrix'Range
                    loop
                        Put "(";
                        Print Integer (Row);
                        Put ",";
                        Print Integer (Col);
                        Put "): ";
                        Matrix(Row, Col) := Real_From_Terminal;
                    end loop;
                end loop;

                if Triangular (Matrix) then
                    Determinant := Determinant_Of_Triangular (Matrix);
                    Put ("Determinant of matrix is ");
                    Print Real (Determinant);
                    New_Line;
                else
                    Put_Line ("Matrix is not triangular.");
                end if;

            end Get_Data_and_Do_Calc;

    begin -- Calculate_Determinant_Of_Triangular_Matrix

        Put_Line ("Enter the dimension of the matrix (for 3 x 3 " &
            "type ""3""");
        Dimension := Integer_From_Terminal;
        Get_Data_and_Do_Calc (Dimension);

    end Calculate_Determinant_Of_Triangular_Matrix;

```

## RATIONALE FOR SOLUTION

Calculate\_Determinant\_Of\_Triangular\_Matrix reads from the terminal the elements of a square matrix, and stores them in a two-dimensional array. In order to declare a matrix of the correct size and then know how many elements to read into it, the dimension of the array must first be determined. The body of the procedure thus reads in the dimension and passes that value to a nested procedure to perform the actual calculations. The array is passed to the previously written function Triangular to determine whether the matrix is triangular. If Triangular is True, then the array is passed to the function Determinant\_Of\_Triangular and the returned value is printed on the screen. If Triangular is not True, an appropriate message appears. Let's take a closer look:

Calculate\_Determinant\_Of\_Triangular\_Matrix uses procedures from Text\_IO, which is made accessible before the procedure definition using with and use, as is package Matrix\_Operations. Objects for the dimension of the matrix, Dimension, and for the determinant of the matrix, Determinant, are declared; the latter is declared in the nested procedure and belongs to type Real of Matrix\_Operations.

The procedure first prints on the terminal a string requesting the dimension of the matrix to be entered. Because the string is too long to fit on one line, two shorter strings are catenated. The string includes quote marks, which are represented in the literal by double quotes. Put\_Line is used because it prints a New\_Line character after the string. Dimension is assigned the value typed into the terminal. Matrix can then be declared in Get\_Data and\_Do\_Calc with index upper bounds of Dimension, passed as a parameter.

The nested procedure prints another string requesting the elements of the matrix, and they are stored in Matrix using a for loop where the loop parameter bounds are given by Matrix'Range. Matrix'Range(2) is not used for the inner loop because both indices of Matrix have the same range. If the user were to enter a dimension of 6, and then type in 37 elements, the last number would be ignored because Real\_From\_Terminal is called only 36 times. And if only 35 elements were entered, the program would just sit and wait for the user to figure out that another was needed. The program prompts the user with the position of the element being entered, which might be very helpful for large matrices. Note that Put is used for the characters in the positional prompt so that they will run together without spaces or new lines.

Once the array Matrix is filled the function Triangular is called. If Triangular is True, Determinant is assigned the value of Determinant\_Of\_Triangular(Matrix), a string "Determinant of matrix is " is printed using Put, and Determinant is printed on the screen. The string is printed using Put so that the value of the determinant will follow on the same line. A space is included at the end of the string so that the determinant does not run into it. If Triangular(Matrix) is False string "Matrix is not triangular." is printed using Put\_Line.

### ALTERNATIVE SOLUTION

```
with Text_IO;           use Text_IO;
with Matrix_Operations; use Matrix_Operations;
procedure Calculate_Determinant_Of_Triangular_Matrix is

    Dimension      : Integer := Integer_From_Terminal;
    Determinant     : Real;
    Matrix : Matrix_Type (1 .. Dimension, 1 .. Dimension);

begin -- Calculate_Determinant_Of_Triangular_Matrix

    for Col in Matrix'Range          -- fill Matrix
    loop
        for Row in Matrix'Range
        loop
            Matrix(Row, Col) := Real_From_Terminal;
        end loop;
    end loop;

    if Triangular (Matrix) then
        Determinant := Determinant_Of_Triangular (Matrix);
        Put ("Determinant is ");
        Print Real (Determinant);
        New_Line;
    else
        Put_Line ("Not triangular.");
    end if;

end Calculate_Determinant_Of_Triangular_Matrix;
```

## DISCUSSION

The alternate solution is very similar to the first solution. It only lacks the additional strings that the first program prints on the terminal, and it no longer requires the nested procedure since Dimension is initialized using an unprompted call to Integer From Terminal. These messages, however, are very useful to the user and contribute to the quality of the program. In the second solution, the user would have to know that the dimension of the matrix had to be entered first, and that the elements of the matrix should be entered going down the column, left to right. And without the prompt in the first program that tells the user the current position in the matrix, the user is much more likely to err when entering large amounts of data. The second program may be used quite easily by the person who wrote it, but the first program could be used by anyone without prior explanation of the code or even knowledge of programming. The effective utilization of strings can be very important.

## EXERCISE 8.5

### OBJECTIVE

To demonstrate arrays of arrays, and arrays of strings.

### TUTORIAL

As explained in Exercise 8.1 the component type of an array type may be any constrained type. An array having an array type component might be declared as follows:

```
type Matrix_Type is array (Integer range 1 .. 4,  
                           Integer range 1 .. 4) of Boolean;  
type Array_Of_Matrices_Type is array (Integer range<>) of Matrix_Type;
```

Here Array\_Of\_Matrices\_Type is an unconstrained one-dimensional array of arrays of type Matrix\_Type. For every value of the index of Array\_Of\_Matrices\_Type there is a distinct array.

One important note concerning arrays of arrays - an array that is a component of another array must be constrained. One COULD NOT write:

```
type Arr_Type is array (Integer range<>) of Real;  
type Array_Of_Array_Type is array (Integer range 1 .. 14) of Arr_Type;
```

An array of strings might be used for a class roster:

```
subtype Pupil_Name_Type is string (1 .. 10);  
type Number_Of_Students is range 1 .. 40;  
type Roster_Arr_Type is array (Number_Of_Students) of Pupil_Name_Type;
```

Roster\_Arr\_Type stores a list of the names of every student in the class. An object declaration,

```
Sixth_Grade_Class_List : Roster_Arr_Type;
```

creates an object for a class list. We could then make the assignments,

```
Sixth_Grade_Class_List(1) := "Alexey   ";  
Sixth_Grade_Class_List(2) := "Ivan     ";  
Sixth_Grade_Class_List(3) := "Dimitri  ";
```



Or, given the declaration

```
Number_Of_Marys : Integer := 0;
```

we count the number of students in the class named Mary using a loop,

```
begin
  for i in Sixth_Grade_Class_List'Range
  loop
    if Sixth_Grade_Class_List(i) = "Mary" then
      Number_Of_Marys := Number_Of_Marys + 1;
    end if;
  end loop;
end;
```

### PROBLEM

A satellite, in circling the earth, takes infrared photographs of land areas. A specific area is photographed from the same position once every day. Write type declarations that will store all of the photographs for a particular area of 10,000 square meters. Each photo is represented as a grid in which each cell contains the infrared intensity of reflectivity (ranging from 0.0 to 1000.0) for one square meter. Then write an object declaration for the photos, totalling 89, taken of an area in Tanzania.

AD-A165 345

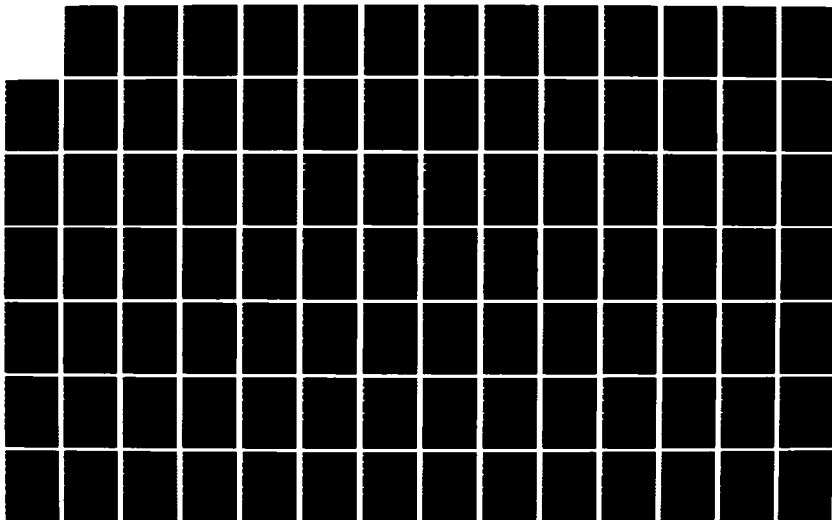
ADA (TRADEMARK) PRIMER(U) SOFTECH INC WALTHAM MA 1986  
DAAB07-83-C-K506

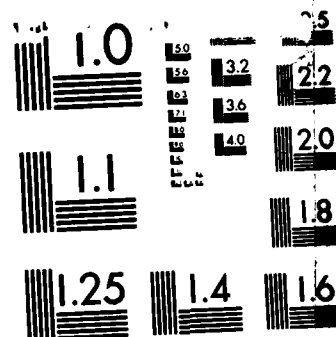
4/5

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A

This page intentionally left blank

## SOLUTION FOR EXERCISE 8.5

### SOLUTION

```
type X_Distance_Type is range 1 .. 100;
type Y_Distance_Type is range 1 .. 100;
type Infrared_Intensity_Type is digits 5 range 0.0 .. 1000.0;
type Graph_Type is array (X_Distance_Type, Y_Distance_Type)
    of Infrared_Intensity_Type;
type Graph_Array_Type is array (Integer range <>) of Graph_Type;

Photos_Of_Tanzania : Graph_Array_Type (1 .. 89);
```

### RATIONALE FOR SOLUTION

The solution declares a two-dimensional array type `Graph_Type` to map the infrared readings of the photographs. The components of `Graph_Type` are of declared type `Infrared_Intensity_Type`. The indices, `X_Distance` and `Y_Distance`, correspond to north-south and east-west scales in meters. The ranges of the indices define `Graph_Type` to be 10,000 meters square. The array type `Graph_Array_Type` is a one-dimensional array all of the maps for a single area. It is unconstrained, allowing any number of photographs. This type may be visualized as a stack of two-dimensional arrays. Finally, `Photos_Of_Tanzania` is declared of type `Graph_Array_Type`. It stores the 89 array maps of an area in Tanzania.

(The notation to reference a particular point on a map looks like:

`Photos_Of_Tanzania(19)(56,20)`

which refers to the point (56,20) on the nineteenth map.)

An array of an array is very suitable for this problem, because it groups in a composite type many related arrays. The maps of Tanzania can easily be compared for new growth, or evaluated for estimations of overall changes.

### ALTERNATIVE SOLUTION

```
type X_Distance_Type is range 1 .. 100;
type Y_Distance_Type is range 1 .. 100;
type Infrared_Intensity_Type is digits 5 range 0.0 .. 1000.0;
type Number_Of_Photos_Type is range 1 .. 89;
type Graph_Type is array (Number_Of_Photos_Type,
    X_Distance_Type,
    Y_Distance_Type) of Infrared_Intensity_Type;

Photos_Of_Tanzania : Graph_Type;
```

## DISCUSSION

The alternate solution declares a single three-dimensional array type for the maps of Tanzania. The first index references the photographs using the declared integer type `Number_Of_Photos_Type`. The second and third indices are for the distances on the maps. The component type is again `Infrared_Intensity_Type`.

This solution is legal, and would probably work well, but the first solution is a better representation of the problem. The first solution allows an unlimited number of photographs for an area, whereas the photo index of the three-dimensional array must be constrained. Further, it stores photographs individually, allowing subprograms to manipulate single photographs.

## **Section 9**

### **RECORD TYPES**

#### **9.1 Record Types/Objects**

#### **9.2 Records of Arrays**

#### **9.3 Arrays of Records**

#### **9.4 Discriminants**



This page intentionally left blank

## EXERCISE 9.1

### OBJECTIVE

To illustrate record types and objects.

### TUTORIAL

There are two kinds of composite types: arrays and records. Unlike arrays, records may have components that are of different types. They are used to express a logical grouping of differently typed data.

The format for a record type declaration is:

```
type Record_Type_Name is
  record
    Component_Name_1 : Component_Type;
    Component_Name_2 : Component_Type;
    .
    .
  end record;
```

Record\_Type\_Name is the name of the type (selected by the user). Component\_Name\_1 and Component\_Name\_2 are the names of components of the record, and Component\_Type is the name of its type. There can be many components. A record type declaration might look like,

```
type Circuit_Rec_Type is
  record
    Current      : Float;
    Resistance    : Float;
    Voltage       : Integer;
    Open          : Boolean;
  end record;
```

Circuit\_Rec Type stores several parcels of data pertaining to a particular circuit. It has four components, Current, Resistance, Voltage, and Open, each of types as indicated. A record component may have any constrained type.

Unlike array types there is no anonymous type for records. One COULD NOT write,

```
Circuit :  
  record  
    Current   : Float;  
    Resistance : Float;  
    Voltage   : Integer;  
    Open      : Boolean;  
  end record;      -- ILLEGAL DECLARATION
```

Two objects of type Circuit\_Rec\_Type could be declared as follows:

```
Circuit_1, Circuit_2 : Circuit_Rec_Type;
```

Its components are referenced using the format:

<u>Record_Name.Component_Name</u>
-----------------------------------

So the Voltage in Circuit\_1 could be set to 100,

```
Circuit_1.Voltage := 100;
```

or the resistance of Circuit\_2 could be assigned the value of the resistance of Circuit\_1 as follows:

```
Circuit_2.Resistance := Circuit_1.Resistance;
```

An object of a record type may be initialized using the format:

<u>Record_Name : Record_Type_Name := Initial_Record_Value;</u>
--

where Initial\_Record\_Value is a record aggregate.

One could initialize an object of type Circuit\_Rec\_Type as follows:

```
Circuit_3 : Circuit_Rec_Type := (8.0, 5.6, 40, False);
```

A record aggregate must be complete. In initialization, there must be a value for every component of the record type.

An individual component of a record may not be declared constant, but an entire record may be. Consider the following record type:

```
type Benzene_Rec_Type is
  record
    Formula      : String (1 .. 4);
    Boiling_Point : Float;
    Freezing_Point : Float;
    Kb           : Float;
    Kf           : Float;
  end record;
```

a constant may be declared of type Benzene\_Rec\_Type as follows:

```
Benzene : constant Benzene_Rec_Type := ("C6H6", 80.2, 5.4, 2.53, 5.12);
```

Benzene is the name of a record having a formula component of "C6H6", a boiling point of 80.2, a freezing point of 5.4, a Kb constant of 2.53, and a Kf constant of 5.12. None of these values change for Benzene, so it makes perfect sense to declare this record a constant. It is thereby prevented from being unintentionally changed.

Record aggregates may be written in either named notation, positional notation, or both. The initial aggregate in the constant declaration for Benzene above is written using positional notation, but it also could have been written with named notation:

```
Benzene : constant Benzene_Rec_Type := (Formula      => "C6H6",
                                         Kb           => 2.53,
                                         Kf           => 5.12,
                                         Boiling_Point => 80.2,
                                         Freezing_Point => 5.4);
```

Note that with named notation the components need not be listed in order.

Both notations may be used, as in the literal,

```
("C6H6", 80.2, Freezing_Point => 5.4, Kb => 2.53, Kf => 5.12);
```

Once named notation has been used in a record aggregate, positional may not again be used. Also, if "others", or | (or) are used for component association, the represented components must all be of the same type.

This page intentionally left blank

### PROBLEM

The storage of several specifications for an automobile is required: its size (length and width), color, weight, maximum velocity, time to accelerate from 0 to 55 mph, mpg for city, mpg for highway, and cubic feet of trunk capacity. Write a type declaration and an initialized object declaration for a typical automobile.

This page intentionally left blank

## SOLUTION TO EXERCISE 9.1

### SOLUTION

```
type Color_type is (Red, Orange, Yellow, Green, Blue, Brown, Silver);
type Real is digits 5;

type Auto_Rec_Type is
  record
    Length      : Real;
    Width       : Real;
    Color       : Color_type;
    Weight      : Integer;
    Maximum_Speed : Real;
    Time_0_To_55 : Real;
    MPG_For_City : Real;
    MPG_For_Highway : Real;
    Trunk_Capacity : Real;
  end record;

Ford_Mustang : Auto_Rec_Type := (Length => 15.7,
                                Width => 9.1,
                                Color => Silver,
                                Weight => 2371,
                                Maximum_Speed => 98.0,
                                Time_0_To_55 => 9.2,
                                MPG_For_City => 23.9,
                                MPG_For_Highway => 29.3,
                                Trunk_Capacity => 8.7);
```

### RATIONALE FOR SOLUTION

The type declaration for the automobile is a record having nine components that correspond to the required specifications. The component Color\_Type is declared an enumeration type for colors, and the weight component is an integer because a fractional part would be insignificant. The other components are of type Real.

The object declaration is for a Ford Mustang car. It is initialized to a record aggregate having typical values for all of the components in Auto\_Rec\_Type.



## ALTERNATIVE SOLUTION

```
type Color_type is (Red, Orange, Yellow, Green, Blue, Brown, Silver);
type Real is digits 5;
```

```
subtype Auto_Length_Type is Real;
subtype Width_Type is Real;
subtype Color_Subtype is Color_Type;
subtype Weight_Type is Integer;
subtype Maximum_Speed_Type is Real;
subtype Time_0_To_55_Type is Real;
subtype MPG_For_City_Type is Real;
subtype MPG_For_Highway_Type is Real;
subtype Trunk_Capacity_Type is Real;
```

```
Auto_Length      : Auto_Length_Type := 15.7;
Width             : Width_Type := 9.1;
Color            : Color_Type := Silver;
Weight           : Weight_Type := 2371;
Maximum_Speed    : Maximum_Speed_Type := 98.0;
Time_0_To_55     : Time_0_To_55_Type := 9.2;
MPG_For_City     : MPG_For_City_Type := 23.9;
MPG_For_Highway  : MPG_For_Highway_Type := 92.3;
Trunk_Capacity   : Trunk_Capacity_Type := 8.7;
```

## DISCUSSION

Another way to code the specifications for the auto is to declare each separately. There is a substantial amount of extra coding. But more importantly, these specifications are not grouped together. In the first solution items to store data for another car can easily be created by declaring another record. (For example, if another record were declared for a Dodge Colt, we could compare Ford Mustang.MPG\_For\_City to Dodge\_Colt.MPG\_For\_City quite easily.) But in the alternate solution each separate item of Dodge\_Colt must be separately created.

The first solution also facilitates the assignment of an entire Auto Rec Type (not to be confused with Auto\_Wreck\_Type!) to another variable of that type. It allows an entire record to be passed to a subprogram, and it prevents the possible confusion of data for two different cars that might result if the second solution were used.

In the first solution the specifications for a certain car are logically grouped, and an Auto\_Rec\_Type has a value in its own right. It is important that the stipulations of the problem are apparent in the chosen data structure.

## EXERCISE 9.2

### OBJECTIVE

To demonstrate records of arrays.

### TUTORIAL

A record component may be of an array type, as in the following example:

```
subtype Parent_Name_Type is String (1 .. 15);
type Student_Record_Rec_Type is
  record
    Full_Name   : String (1 .. 20);
    Parent_Name : Parent_Name_Type;
  end record;
```

Student\_Record\_Rec\_Type is declared a type for a record that stores a student's name and his or her parent's name. Notice that the type of a string component may be declared in the record, or it may be previously declared. The subtype of Full\_Name is declared in the record, but Parent\_Name is of declared subtype Parent\_Name\_Type.

However, a record component of an anonymous array type cannot be declared in the record. In other words, the following construct would NOT be allowed:

```
type Circuit_Panel_Rec_Type is
  record
    Switch_1 : array (1 .. 10) of Boolean;
    Switch_2 : array (1 .. 10) of Boolean;
  end record; -- ILLEGAL
```

The above record would be written:

```
type Switch_Arr_Type is array (1 .. 10) of Boolean;
type Circuit_Panel_Rec_Type is
  record
    Switch_1 : Switch_Arr_Type;
    Switch_2 : Switch_Arr_Type;
  end record;
```

Further, a record component may not be of an unconstrained array type without including a range constraint. For example, in the following record type,

```
type Vehicle_Rec_Type is
  record
    Mass      : Real;
    Velocity  : Matrix_Arr_Type (1 .. 2);
    Movement  : Matrix_Arr_Type (1 .. 3);
  end record;
```

where

```
type Matrix_Arr_Type is (Integer range < >) of Real;
```

the declaration of the components Velocity and Movement must contain the given range constraint because Matrix\_Arr\_Type is unconstrained.

### PROBLEM

Write type and object declarations that, for a given patient, will store the readings taken during an operation. The values required are: the name of the patient, the blood pressure every 15 seconds during the operation, and the heart rate every minute during the operation. Assume that the operation is two hours long. (The heart rate is an integer value, and the blood pressure is a pair of integers.)

This page intentionally left blank

## SOLUTION TO EXERCISE 9.2

### SOLUTION

```
Length_Of_Operation : Integer := 120;           -- in minutes

type Heart_Rate_Arr_Type is array (1 .. Length_Of_Operation)
  of Integer;

type Blood_Pressure_Reading_Rec_Type is
  record
    Systolic_Part : Integer;
    Diastolic_Part : Integer;
  end record;
type Blood_Pressure_Arr_Type is array (1 .. 4 * Length_Of_Operation)
  of Blood_Pressure_Reading_Rec_Type;           -- 4 times a min.

type Operation_Readings_Rec_Type is
  record
    Patient_Name : String (1 .. 20);
    Heart_Rate : Heart_Rate_Arr_Type;
    Blood_Pressure : Blood_Pressure_Arr_Type;
  end record;

Operation_Readings : Operation_Readings_Rec_Type;
```

### RATIONALE FOR SOLUTION

A record called `Operation_Readings` is declared to store the readings taken during the operation. It has three components: the name of the patient, the heart rate, and the blood pressure.

The name of the patient `Patient_Name` is of type `String`. It has twenty characters. The heart rate `Heart_Rate` is of declared array type `Heart_Rate_Arr_Type`, which is one-dimensional. It has `Length_Of_Operation`, or 120, components. Each component is of predefined type `Integer`, corresponding to the pulse.

The blood pressure component of `Operation_Readings`, `Blood_Pressure`, is a declared array type `Blood_Pressure_Arr_Type`, which is also one-dimensional. There are four times as many components in `Blood_Pressure_Arr_Type` as in `Heart_Rate_Arr_Type` because the blood pressure is taken four times a minute. Each component of `Blood_Pressure_Arr_Type` is a pair of integers. That pair is represented by a record type `Blood_Pressure_Reading_Rec_Type`, which has just two integer components.

### ALTERNATIVE SOLUTION

```
Length_Of_Operation : Integer := 120;           -- in minutes

type Heart_Rate_Arr_Type is array (1 .. Length_Of_Operation)
  of Integer;
Heart_Rate : Heart_Rate_Arr_Type;

type Blood_Pressure_Reading_Rec_Type is
  record
    Systolic_Part : Integer;
    Diastolic_Part : Integer;
  end record;
type Blood_Pressure_Arr_Type is array (1 .. Length_Of_Operation * 4)
  of Blood_Pressure_Reading_Rec_Type;           -- 4 times a min.

Blood_Pressure : Blood_Pressure_Arr_Type;

Patient_Name : String (1 .. 20);
```

### DISCUSSION

The alternate solution declares all of the types of the record components in the first solution: Patient\_Name, Heart\_Rate, and Blood\_Pressure, but it does not declare a record to associate them to one another. The alternate solution requires that, for every patient, three items must be declared. But in the first solution, one could simply create a record for every patient. And the readings taken during the operation for that patient would be logically grouped together. For instance, if we create a record for patient number 23,

```
Operation_Readings_Patient_23 : Operation_Readings_Rec_Type;
```

we can refer to his or her blood pressure two minutes into the operation as follows:

```
Operation_Readings_Patient_23.Blood_Pressure(8)
```

But the alternate solution would require a separate declaration for the blood pressure readings of patient number 23, in fact for all three items:

```
Patient_23_Name : Patient_Name;
Patient_23_Heart_Rate : Heart_Rate;
Patient_23_Blood_Pressure : Blood_Pressure;
```

and the blood pressure after two minutes could then be accessed,

```
Patient_23_Blood_Pressure(8)
```

The use of the record type is advantageous for this problem. It groups together the various readings taken during the operation, thereby making this data simpler, easier to understand and read, and easier to handle.



This page intentionally left blank

## EXERCISE 9.3

### OBJECTIVE

To demonstrate arrays of records.

### TUTORIAL

As explained in Exercise 8.1, the component of an array may be of any constrained type. Arrays of records can be particularly useful in storing large amounts of data. Consider the following declarations:

```
type Real is digits 4;

type Cartesian_Point_Rec_Type is
  record
    X : Real;
    Y : Real;
    Z : Real;
  end record;
type Cartesian_Graph_Arr_Type is array (Integer range < >)
  of Cartesian_Point_Rec_Type;
Cartesian_Graph : Cartesian_Graph_Arr_Type (1 .. 5000);
```

Cartesian\_Point\_Rec\_Type is declared a record type for a single point in a three-dimensional area. Its components are the X, Y, and Z real coordinates of the point. For example, the point (1.0, 2.0, 3.0) could be declared as follows:

```
Cartesian_Point : Cartesian_Point_Rec_Type := (X => 1.0,
                                                Y => 2.0,
                                                Z => 3.0);
```

Cartesian\_Graph\_Arr\_Type is an array of Cartesian\_Point\_Rec\_Type. Cartesian\_Graph is an object of type Cartesian\_Graph\_Arr\_Type. It stores 5000 points of a three dimensional graph. One could create a point (1.0, 2.0, 3.0) within Cartesian\_Graph this way:

```
Cartesian_Graph(1) := (1.0, 2.0, 3.0);
```

Because the components of Cartesian\_Point\_Rec\_Type are all the same, a Cartesian graph could also be created using only array types:

```
type Components_Type is (X, Y, Z);
type Cartesian_Point_Arr_Type is array (Components_Type) of Real;
type Cartesian_Graph_Arr_Type_A is array (Integer range < >)
  of Cartesian_Point_Arr_Type;
Cartesian_Graph_A : Cartesian_Graph_Arr_Type_A;
```

Instead of a record type as above, Cartesian\_Point\_Arr\_Type here is an array with an index of declared enumeration type Components\_Type. The rest is the same as the other declarations above. Note that the notation for referencing the X coordinate of the eighth point in the Cartesian\_Graph is

Cartesian\_Graph(8).X

whereas in Cartesian\_Graph\_A the notation is

Cartesian\_Graph\_A (8)(X)

This graph could be represented using either an array of a record type or an array of an array type only because the type of the three coordinates are the same. In most circumstances, a record would be needed to accommodate components of different types, as demonstrated in the problem.

### PROBLEM

Write the necessary declarations to represent the periodic table. The values required are the atomic number (integer 1 - 103), the atomic weight (real 0.0 - 300.0), the symbol of the element, and the block (S, P, D, or F).

This page intentionally left blank

## SOLUTION TO EXERCISE 9.3

### SOLUTION

```
type Atomic_Number_Type is range 1 .. 103;
type Atomic_Weight_Type is digits 8 range 0.0 .. 300.0;
type Block_Type is (S, P, D, F);

type Element_Rec_Type is
  record
    Element_Symbol : String (1 .. 2);
    Atomic_Weight  : Atomic_Weight_Type;
    Block          : Block_Type;
  end record;

type Periodic_Table_Arr_Type is array (Atomic_Number_Type)
  of Element_Rec_Type;

Periodic_Table : constant Periodic_Table_Arr_Type
  := (1 =>("H ", 1.0080, S),
      2 =>("He", 4.003, P),
      .
      .
      .
      103 =>("Lr", 257.0, F));
```

### RATIONALE FOR SOLUTION

The periodic table here is represented as an array of elements, where each element is a record having three components. The first is declared in the record type declaration, and is a string of two characters for the element symbol. The second component is for the atomic weight of the element, and is of declared type `Atomic_Weight_Type`, a floating point type with a range from 0.0 to 300.0. The third element specifies the block of the element, and is of declared enumeration type `Block_Type`.

`Periodic_Table_Arr_Type` is declared an array type with an index of type `Atomic_Number_Type`. Each component of `Periodic_Table_Arr_Type` is a record for an element. The atomic number is a good index because it has an integer value, and there is an element for every value between 1 and 103. It is also convenient when using the table to refer to the element having atomic number of 98, or whatever.

`Periodic_Table` is declared a constant object of type `Periodic_Table_Arr_Type`. It is constant because it is a look-up table of invariant values. There is very little likelihood that the atomic weight of Hydrogen will change. The initial record aggregate is not completely written here because it is too long.

Note that an individual element could not be represented by an array because the atomic weight, the symbol, and the block types are all different. An array of a record type is an excellent way to handle the given problem. To access the name of the element having an atomic number of 32 one would write:

```
Periodic_Table(32).Element_Symbol
```

And one could search the table for the atomic weight of an element as follows:

```
function Atomic_Weight_Of (Symbol: String (1..2))
    return Atomic_Weight_Type is
    I : Atomic_Number_Type := 1;
begin
    while I <= Periodic_Table'Last and then
        Periodic_Table(I).Element_Symbol /= Symbol
    loop
        I := I + 1;
    end loop;
    if I <= Periodic_Table'Last then
        return Periodic_Table(I). Atomic_Weight;
    else
        return 0.0; -- a nonexistent weight
    end if;
end Atomic_Weight_Of;
```

To search for the atomic weight of the element having the symbol "Sn" one would invoke the function as follows: Atomic\_Weight\_Of ("Sn")

Note that one could also create a record for each element containing components for the atomic weight, the block, and the atomic number, and then declare an array of such records that is indexed by the symbols (using an enumeration type). One could then refer to the atomic number of Germanium as follows:

```
Periodic_Table(Ge).Atomic_Number
```

Whether the records are indexed by atomic numbers or element symbols is a matter of preference.

## ALTERNATIVE SOLUTION

```
type Real is digits 5;
type Element_Categories_Type is (Atomic_Weight, Atomic_Number, Block);
type Element_Arr_Type is array (Element_Categories_Type) of Real;
type Element_Symbol is (H, He, Li, Be, B, C, N, O, F, Ne, Na, Mg, Al,
                        Si, P, S, Cl, Ar, K, Ca, Sc, Ti, V, Cr, Mn,
                        Fe, Co, Ni, Cu, Zn, Ga, Ge, As, Se, Br, Kr,
                        Rb, Sr, Y, Zr, Nb, Mo, Tc, Ru, Rh, Pd, Ag, Cd,
                        In_E, Sn, Sb, Te, I, Xe, Cs, Ba, La, Ce, Pr,
                        Nd, Pm, Sm, Eu, Gd, Tb, Dy, Ho, Er, Tm, Yb,
                        Lu, Hf, Ta, W, Re, Os, Ir, Pt, Au, Hg, Tl, Pb,
                        Bi, Po, At_E, Rn, Fr, Ra, Ac, Th, Pa, U, Np,
                        Pu, Am, Cm, Bk, Cf, Es, Fm, Md, O, Lr);

type Periodic_Table_Type is array (Element_Symbol)
  of Element_Arr_Type;
Periodic_Table : constant Periodic_Table_Arr_Type
  := (H => (1.0080, 1.0, 1.0),
      He => (4.003, 2.0, 2.0),
      .
      .
      .
      Lr => (257.0, 103.0, 4.0));
```

## DISCUSSION

The alternate solution uses an array type to store the values of the periodic table. The array index is an enumeration type Element\_Symbol, which is a list of the symbols of all of the elements. (Notice that In is In\_E and At is At\_E because "in" and "at" are reserved words.) Each component of the array is an array of type Element\_Arr\_Type, which has three components which are indexed by an enumeration type. The components are Atomic\_Weight, Atomic\_Number, and Block. Each component is of type Real. To refer to the Block of Mercury, one would write,

Periodic\_Table(Hg)(Block)

and the returned value would be a real number. For Blocks the number 1.0, 2.0, 3.0, or 4.0 refers to the S, P, D, or F blocks.

This solution is really quite messy. Using array types requires that every component be of the same type, Real in this case. Atomic weights are best described using real values, but neither atomic numbers nor block types have real values. Consequently, the blocks are represented using 1.0, 2.0, 3.0, and 4.0, and the atomic numbers, too, have an unnecessary decimal part. The values returned by the table are inappropriate, even difficult to interpret.



This page intentionally left blank

## EXERCISE 9.4

### OBJECTIVE

To introduce records with discriminants.

### TUTORIAL

Records are used to group together logically related data, forming a single structure out of components of different types. In some cases, the logical structure includes certain dependencies among the components; for instance, the length of an array component might depend on the value of some discrete-typed component, or one component's value might determine whether certain others contain significant information. The set of components needed by each object of the type would thus vary according to its purpose. For example, a record type representing various sensors throughout a building might need a Temperature component for a heat sensor object but not for a dust sensor object, whereas the dust sensor would need only a Dust Level component. Another situation would be a record defining a text Buffer type containing a string component, where the string has no predefined maximum length and will vary in length from one buffer object to the next.

Discriminants allow for such variation within a single record type, playing a role similar to that of subprogram parameters. A discriminant may serve as part of the definition of other record components, as in providing the upper index bound for an array component; or it might be used to provide a variant part, specifying alternative lists of components for each possible value of the discriminant, as discussed below. This exercise presents the syntax for discriminants and variant records and illustrates how these structures might be used.

The declaration of a record type with discriminants takes the form:

```
type Record_Type_Name (Discriminant_Specification { ;  
    Discriminant_Specification } ) is  
    record  
        .      -- component declarations  
        .  
    end record;
```

where the form of a discriminant specification is as follows:

```
Identifier { , Identifier } : Discrete_Type_Name  
    [ := Default_Initial_Value ]
```

Note that elements surrounded by braces may appear zero or more times, and square brackets indicate optional parts. The element Discrete\_Type\_Name indicates that, like case expressions (Exercise 5.2), discriminants must belong to a discrete type (enumeration or integer).

One use for discriminants is in creating record types with different forms, called variants, that allow us to better reflect the logical structure of the data. For example, consider the problem of representing library card catalog information. All cards show the title, author, Dewey decimal number, subject and year of the work; in addition, the card for a book should show the publisher and the number of pages, while the card for an article or paper should hold the name, issue, and starting page number of the periodical or journal containing the work.

The declarations for the card catalog entry might be as follows:

```
subtype Title_Subtype      is String (1 .. 50);
subtype Author_Subtype     is String (1 .. 30);
subtype Subject_Subtype    is String (1 .. 20);
subtype Publisher_Subtype  is String (1 .. 30);

subtype Year_Subtype       is Integer range 1800 .. 2100;
subtype Dewey_Decimal_Subtype is Float range 0.0 .. 999.999;

type Item_Mode_Type is (Book, Article, Paper);

type Card_Catalog_Type is
  record
    Title_Part      : Title_Subtype;
    Author_Part     : Author_Subtype;
    Subject_Part    : Subject_Subtype;
    Year_Part       : Year_Subtype;
    Reference_Number : Dewey_Decimal_Subtype;
    Item_Mode       : Item_Mode_Type;
    Publisher_Part  : Publisher_Subtype;
    Number_of_Pages : Natural;
    Periodical_Part : Title_Subtype;
    Issue_Number    : Natural;
    Page_Number     : Natural;
  end record;
```

The above record, however, does not reflect the fact that the last three components are meaningless if the entry is a book, or that Publisher Part and Number of Pages are extraneous for articles and papers. Since the choice of components depends on the value of the Item Mode component, we make it a discriminant; we can thus use it to determine the appropriate record structure. By adding a discriminant to the record type, we can allow both a book form and an article/paper form, with components to hold the necessary information in each case:

```

type Card_Catalog_Type (Item_Mode : Item_Mode_Type) is
  record
    Title_Part       : Title_Subtype;
    Author_Part      : Author_Subtype;
    Subject_Part     : Subject_Subtype;
    Year_Part        : Year_Subtype;
    Reference_Number : Dewey_Decimal_Subtype;
    case Item_Mode is
      when Book =>
        Publisher_Part : Publisher_Subtype;
        Number_Of_Pages : Natural;
      when Article | Paper =>
        Periodical_Part : Title_Subtype;
        Issue_Number    : Natural;
        Page_Number     : Natural;
    end case;
  end record;

```

Each Card\_Catalog\_Type object still has a component named Item Mode, but it is specified as a discriminant. The value of this component will now determine whether the object has a Publisher Part and a Number of Pages, or rather a Periodical Part, an Issue Number, and a Page Number. The other components are independent of this discriminant, so all objects of Card\_Catalog\_Type will have a Title Part, an Author Part, a Subject Part, a Year Part, and a Reference Number.

The variant part of a record is structurally similar to a case statement (see Exercises 5.2 and 7.2). The general form of a variant record is shown below, followed by the form of the variant part:

Variant record form:

```
type Record_Type_Name (Discriminant_Specifications) is
  record
    Component_declaration;
    Variant_Part;
  end record;
```

Variant part form:

```
case Discriminant_Name is
  when Choice { | Choice } =>
    Component_List;
  when Choice { | Choice } =>
    Component_List;
  .
  . -- and so on until all possible discriminant values
  . -- are accounted for
end case;
```

Note that the variant part must appear last in the record, after all other component declarations.

As in a case statement, the choice others may be used as the last option in a variant part. There must be one set of components for each possible value of the discriminant, just as there must be one alternative for each possible value of the case expression. A null component must be specified if the set of components for a particular discriminant value is empty:

```
case Discriminant_Name is
  when Value_1 =>
    Component : Component_Type;
  when others =>
    null;
end case;
```

Only one variant part may be included in a given record declaration. Variants may be nested, however; that is, the component list of one branch of a variant part may itself contain a variant part, calling upon a different discriminant.

Another use for discriminants is in controlling the length of arrays inside records. Recall that an unconstrained array type may not be used to define a record component unless an index constraint is supplied, thus limiting the length of a record's array component to some arbitrary maximum value. Alternatively, discriminants may be used to specify the array bounds, so that different records in the record type may have array-valued components of different sizes.

Consider a modification to the library catalog example above: in addition to the subject shown on the particular card, each card is to have a list of cross-references to other subjects, each of which has a card for the entry in question. Since the number of cross-references is not known in advance, it would be advantageous to use an unconstrained array type for the list:

```
type Cross_Reference_List_Type is
    array ( Natural range <> ) of Subject_Subtype;
```

To be used as a record component type, however, the list must be constrained; we thus add a Number\_of\_References discriminant and use that component to constrain the array:

```
type Card_Catalog_Type (Item_Mode : Item_Mode_Type;
                        Number_of_References : Natural) is
    record
        Title_Part      : Title_Subtype;
        Author_Part     : Author_Subtype;
        Subject_Part    : Subject_Subtype;
        Year_Part       : Year_Subtype;
        Reference_Number : Dewey_Decimal_Subtype;
        Cross_References : Cross_Reference_List_Type
                        (1.. Number_of_References);
        case Item_Mode is
            when Book =>
                Publisher_Part : Publisher_Subtype;
                Number_Of_Pages : Natural;
            when Article | Paper =>
                Periodical_Part : Title_Subtype;
                Issue_Number    : Natural;
                Page_Number     : Natural;
        end case;
    end record;
```

A common application of discriminants used as array bounds is in allowing strings of varying lengths, as in the text buffer situation mentioned above. For another example, suppose each book, article, and paper in the card catalog has an associated piece of text describing the subject matter. To allow these descriptors to be of any length, we might define a Contents\_Descriptor\_Type record as follows:

```

type Contents_Descriptor_Type (Length : Natural) is
  record
    Text_Part : String (1 .. Length);
  end record;

```

To add a Contents\_Descriptor component to each card catalog entry, we must also add a third discriminant, Contents\_Descriptor\_Length, that will in turn be used to specify a value for the discriminant of the Contents\_Descriptor\_Type component. Any object of a record type with discriminants must be provided with a value for each discriminant, either by default (discussed below) or by an explicit constraint. With the addition of a Contents\_Descriptor component and the associated discriminant, the card catalog entry appears as follows:

```

type Card_Catalog_Type (Item_Mode : Item_Mode_Type;
                        Number_of_References : Natural;
                        Contents_Descriptor_Length : Positive ) is
  record
    Title_Part      : Title_Subtype;
    Author_Part     : Author_Subtype;
    Subject_Part    : Subject_Subtype;
    Year_Part       : Year_Subtype;
    Reference_Number : Dewey_Decimal_Subtype;
    Cross_References : Cross_Reference_List_Type
                        (1 .. Number_of_References);
    Contents_Descriptor : Contents_Descriptor_Type
                        (Contents_Descriptor_Length);
    case Item_Mode is
      when Book =>
        Publisher_Part : Publisher_Subtype;
        Number_Of_Pages : Natural;
      when Article | Paper =>
        Periodical_Part : Title_Subtype;
        Issue_Number    : Natural;
        Page_Number     : Natural;
    end case;
  end record;

```

The addition of a Contents\_Descriptor component thus illustrates a third use for discriminants, in providing a constraint for a component that is itself a record with discriminants.

As with subprogram parameter associations, discriminant constraints may be written in named, positional, or mixed notation; recall that if mixed notation is used, all positional constraints must come first. An object of Contents\_Descriptor\_Type could thus be declared in either of the following forms:

```
Contents_Descriptor_1 : Contents_Descriptor_Type (80);
```

```
Contents_Descriptor_2 : Contents_Descriptor_Type (Length => 80);
```

Wherever it is used, a discriminant must appear as a full expression in itself; it may not be part of a larger expression. This restriction holds for all uses of discriminants: as a choice in a variant part; in providing bounds for an array component; as a constraint for a component that is itself a record with discriminants; and as the initial value of components. This final use has not been discussed here, but a typical application might look like this:

```
type Some_Record_Type (Initial_Count : Positive) is
  record
    Count_Part : Positive := Initial_Count;
    .          -- other component declarations
  end record;
```

Like any other record component, a discriminant may be given a default value. Unlike defaults for other components, however, default values must be provided for all of a record type's discriminants or for none of them. With other components, the declaration may specify defaults for an arbitrary subset of the components. We could provide defaults for the discriminants of Card\_Catalog\_Type as follows:

```
type Card_Catalog_Type (Item_Mode : Item_Mode_Type := Book;
                        Number_of_References : Natural := 1;
                        Contents_Descriptor_Length : Positive := 80 )
is
  . . .
```

If no default discriminant values are provided in a record type declaration, every object in the type must be declared with a constraint. (This restriction is similar to that regarding objects of an unconstrained array type.) Objects of record types with default discriminants may be declared with or without discriminant values; the defaults are used only if no other values are given. Any object declared with a discriminant constraint is said to be constrained, and its discriminants may never take on values other than those supplied in the object declaration. For example, in the declarations of Contents\_Descriptor\_1 and Contents\_Descriptor\_2 above, both objects are constrained, with a Length component permanently set to the value 80.



Alternatively, an object of a type with default discriminant values may be declared with no discriminant constraint; its discriminants originally take on the default values, but may later be changed, as described below. Such objects are said to be unconstrained. The attribute 'Constrained may be applied to an object of a record type with discriminants, returning True if the object is constrained and False if it is not.

The value of the discriminant of an unconstrained object may not be changed independently of the other components; instead, every component of the record must be reassigned, using either a complete record aggregate or another record object of the same type. The definition of other components may depend on the value of the discriminant, so changing the discriminant alone could result in inconsistent component values. Thus a discriminant may never appear on the left-hand side of an assignment statement or as an out or in out actual parameter; only a whole record variable or a record aggregate assignment may be used, even if no other components are altered. (In a positional aggregate for a record with discriminants, the values for the discriminants appear first, in the same order as in the specification.)

The statements below illustrate how Contents\_Descriptor\_Type might be redeclared using a default Length value so that an unconstrained Contents\_Descriptor object may be declared:

```

type Contents_Descriptor_Type_2 (Length : Positive := 10) is
  record
    Text_Part : String (1 .. Length);
  end record;

Contents_Descriptor_3 : Contents_Descriptor_Type_2;
                        -- Contents_Descriptor_3.Length = 10

. . .

Contents_Descriptor_3.Text_Part := "Hi, world.";
Contents_Descriptor_3 := (Length    => 8,
                        Text_Part => "Goodbye.");
                        -- Now Contents_Descriptor_3.Length = 8

Contents_Descriptor_3.Length := 7;                                --**ILLEGAL**

```

Note that in the second assignment to Contents\_Descriptor\_3, the change in discriminant value is reflected in the text component value; the text string is now 8 characters long rather than 10. This change conforms to the consistency rules associated with discriminants, in that record aggregates must be consistent with discriminant values, as must

references to components. In addition, the discriminants of an object must always have values, and those values may not be changed independently of the rest of the record, as discussed above. The attempt to assign the value 7 to Contents\_Descriptor\_3.Length is thus illegal.

Subtypes of record types with discriminants may be used to specify the subset of record objects with a particular discriminant value. For example, the declaration

```
subtype Ten_Char_Contents_Descriptor_Subtype is
    Contents_Descriptor_Type (Length => 10);
```

defines a subtype of Contents\_Descriptor\_Type consisting of all descriptors of length 10. Membership tests can then be used to see if objects of the base type belong to the particular subtype:

```
if Contents_Descriptor_1 in Ten_Char_Contents_Descriptor_Subtype then
    . . .
end if;
```

Subtypes are particularly useful when dealing with record types with many discriminants.

### PROBLEM

Write the declarations needed to define a record type representing points in two-dimensional space, such that either Cartesian or Polar coordinates may be used. A Cartesian point needs an X-Coordinate and a Y-Coordinate, both of which are real numbers, whereas a point in Polar form is represented by an angle (in degrees) and a magnitude (a positive real number). Keep in mind that it might be necessary to convert any given point from one representation to the other, so it should be possible to declare unconstrained objects of the type.

Using this point type, define a `Point_Set_Type` that provides a means to group together an arbitrary number of points, based on the variable-length array types discussed in the Tutorial. An object of such a type should be a record with a component belonging to an unconstrained array type, with a discriminant providing the upper bound for the array. Declare and initialize three Cartesian points (using any values); then declare an object of `Point_Set_Type` to hold the values of those points.

## SOLUTION TO EXERCISE 9.4

### SOLUTION

```
subtype Angle_Subtype is Float range 0.0 .. 360.0;
subtype Positive_Float is Float range 0.0 .. Float'Last;

type Coordinate_System_Type is (Cartesian, Polar);

type Point_Type (Coordinate_System : Coordinate_System_Type := Polar)
is record
  case Coordinate_System is
    when Cartesian =>
      X_Coordinate, Y_Coordinate : Float;
    when Polar =>
      Angle_Part      : Angle_Subtype;
      Magnitude_Part : Positive_Float;
  end case;
end record;

type Point_Array_Type is array (Positive range<>) of Point_Type;

type Point_Set_Type (Number_of_Points : Natural := 2 ) is
record
  Point_Set : Point_Array_Type (1 .. Number_of_Points);
end record;

Point_1 : Point_Type := (Cartesian, 10.5, -2.3);
Point_2 : Point_Type := (Cartesian, 1.4, 12.85);
Point_3 : Point_Type := (Cartesian, 21.0, 28.0);

Point_Set_1 : Point_Set_Type := (3, (Point_1, Point_2, Point_3));
```

## RATIONALE FOR SOLUTION

Two-dimensional points are represented here by the variant record type `Point_Type`, in which all components depend on the value of the `Coordinate_System` discriminant. `Coordinate_System` belongs to the enumeration type `Coordinate_System_Type`, which is a discrete type and is therefore a legal type for the discriminant. This type consists of the two values `Cartesian` and `Polar`; the value assigned to the discriminant determines the appropriate components needed to represent a point in the given system.

The discriminant is arbitrarily given the default value `Polar`, so that unconstrained objects of type `Point_Type` may be declared. Such objects have the flexibility to be converted from one coordinate system to the other, as specified by the Problem. For example, an unconstrained point object might originally be assigned the values `(Cartesian, 0.0, 1.0)`, used in that form for certain computations, then passed to a conversion routine that reassigns to it the corresponding `Polar` values `(Polar, 90.0, 1.0)`, facilitating some other computation.

There are two components in each branch of the variant part of `Point_Type`: `X Coordinate` and `Y Coordinate` for the `Cartesian` variant; `Angle_Part` and `Magnitude_Part` for the `Polar` variant. Both `Cartesian` components are of type `Float`. The `Polar` component `Magnitude_Part` belongs to the subtype `Positive_Float`, which ranges from `0.0` to `Float'Last` (the `'Last` attribute returns the largest number in the type, according to the implementation). `Angle_Part` belongs to `Angle_Subtype`, a `Float` subtype limited to the range `0.0` to `360.0`.

`Point_Set_Type` is a record type with a discriminant of type `Natural` and a `Point_Set` component of the unconstrained array type `Point_Array_Type`. The discriminant `Number_of_Points` is used to provide the upper bound for the array component `Point_Set`. The elements of this array are of type `Point_Type`; since this type has a default discriminant value, no constraint is needed for the array elements themselves. `Number_of_Points` is also provided with a default, making it possible to declare point sets that may vary in size, just as the points themselves may change from one coordinate representation to the other.

The three `Point_Type` variables are declared as unconstrained objects; each is initialized with a discriminant value of `Cartesian` and appropriate coordinate values. `Point_Set_1` is also an unconstrained object; it is initialized by a positional record aggregate, with the first element specifying a discriminant value of `3`, and the following array aggregate assigning the values of the three points to the array component of the record. Note that the discriminant value is consistent with the number of values assigned to the array component.

### ALTERNATIVE SOLUTION

```
subtype Angle_Subtype is Float range 0.0 .. 360.0;
subtype Positive_Float is Float range 0.0 .. Float'Last;

type Point_Type is
  record
    X_Coordinate, Y_Coordinate : Float;
    Angle_Part : Angle_Subtype;
    Magnitude_Part : Positive_Float;
  end record;

type Point_Array_Type is array (Positive range<>) of Point_Type;

type Point_Set_Type (Number_of_Points : Natural := 2 ) is
  record
    Point_Set : Point_Array_Type (1 .. Number_of_Points);
  end record;

Point_1 : Point_Type := (10.5, -2.3, 0.0, 0.0);
Point_2 : Point_Type := (1.4, 12.85, 0.0, 0.0);
Point_3 : Point_Type := (21.0, 28.0, 53.13, 35.0);

Point_Set_1 : Point_Set_Type := (3, (Point_1, Point_2, Point_3));
```

### DISCUSSION

In the alternate solution, Point\_Type is no longer a variant record type. The discriminant has been removed, eliminating the need for a Coordinate\_System\_Type. Each object of the type now has four components, although in general only the two appropriate to the coordinate system being used will contain meaningful values.

The advantage to this structure is that it is possible to hold both the Cartesian and the Polar representation of a given point in a single object, illustrated by Point\_3 above, eliminating the need for conversion routines. Note, however, that there is no guarantee that the values of the angle and magnitude components will always correspond to those of the Cartesian components; in Point\_1 and Point\_2 they clearly do not. Also, when positional notation is used, the meaning of each value in the aggregate is not inherently obvious; it is not clear that the first two values form a pair, as do the last two.

This solution also makes it impossible to declare a subtype containing only points in one of the two forms, whereas in the original solution the following subtype might be declared:

```
subtype Polar_Point_Type is Point_Type
    (Coordinate_System => Polar);
```

Polar\_Point\_Type consists of all Point\_Type objects with a discriminant value of Polar; that is, all points written in Polar coordinates.

The declarations for Point\_Array\_Type and Point\_Set\_Type are unchanged in the alternative solution. The only modifications necessary in the object declarations are the removal of the discriminant values from the aggregates used to initialize the three points, since the type no longer has a discriminant, and the addition of two component values to complete each aggregate, since each record now has four components instead of two. This representation is thus less accurate and less informative than that of the original solution; record aggregates here do not reflect the coordinate system being used, and they must contain four component values, even if only two are meaningful for a given application.

## **Section 10**

### **ACCESS TYPES**

#### **10.1 Access Types and Allocators**



This page intentionally left blank

## EXERCISE 10.1

### OBJECTIVE

To introduce access types and allocators, and to discuss a common application of access values.

### TUTORIAL

An object of an access type is an indirect reference or "pointer" to another object. Pointers are often found in assembly language programs, when one variable stores the address of another. As this exercise demonstrates, access types introduce alternate ways to organize data and increase the flexibility of data structures. In the illustration of memory space below,



the left box represents an access value; it is said to point to, or designate, the value in the right box. The special access value null is used to denote a pointer that does not designate any object. Since a single access value may point to an object of arbitrary size, the use of access types may save both time and space, making programs more efficient.

In certain respects, access values are similar to addresses. They provide a reference to another object; they may be used as values themselves in some contexts; and two variables containing the same access value provide indirect references to the same object. In contrast to addresses, however, access values may not be used to designate arbitrary memory locations (only objects of a particular type), and access values are not numbers, so no arithmetic or relational operators can take operands of an access type.

An access type declaration takes the following form:

```
type Access_Type_Name is
    access Designated_Subtype_Name [Constraint];
```

The Designated\_Subtype\_Name and optional Constraint specify the subtype of variables to which the access type is to refer. There are no particular limitations on the designated type of an access type; note, however, that an access value may only point to an object of the type specified.

The following declarations define an access type for objects that point to integer values in the range 0 to 500 and create two objects of that access type:

```
type Integer_Pointer_Type is access Integer range 0 .. 500;
Integer_Pointer_1, Integer_Pointer_2 : Integer_Pointer_Type;
```

Integer\_Pointer\_1 and Integer\_Pointer\_2 are access objects containing access values; they are automatically initialized to null, since no other value is given in the declaration. The two objects could equivalently be declared in two separate declarations:

```
Integer_Pointer_1 : Integer_Pointer_Type;
Integer_Pointer_2 : Integer_Pointer_Type;
```

In memory, the result of either form of declaration may be depicted as follows:

Integer_Pointer_1:	null
Integer_Pointer_2:	null

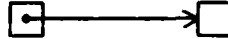
As suggested above, access types may be defined to designate any of a wide variety of object types. For example, suppose we have the following type and subtype declarations:

```
subtype Small_Float_Subtype is Float range 0.0 .. 100.0;
type Color_Type is (Red, Orange, Yellow, Green, Blue, Violet);
type Bit_Sequence_Type is array (Natural range <>) of Boolean;
type Car_Type is
  record
    Make_Part      : String (1 .. 20);
    Weight_Part    : Positive;
    Color_Part     : Color_Type;
    MPG_Part       : Small_Float_Subtype;
  end record;
```

We can then declare access types to designate objects of each:

```
type Small_Float_Pointer_Type is access Small_Float_Subtype;  
type Color_Pointer_Type is access Color_Type;  
type Bit_Sequence_Pointer_Type is access Bit_Sequence_Type;  
type Car_Pointer_Type is access Car_Type;
```

Consider again our simple diagram of a pointer and its designated object:



In this diagram of memory, the right-hand box represents an allocated variable. Unlike declared variables, allocated variables are created via the process of dynamic allocation, which occurs during program execution. At that time, memory space for the object is set aside, and an access value pointing to the object is made available to the program. That access value provides the only means to refer to the allocated variable. In contrast, declared variables are created when the program is compiled, and are referred to directly, by name. Note that allocated variables and declared variables form two distinct classes: An access type may never point to a declared variable, and an allocated variable may be referred to only by an access value.

A process often known as dereferencing is used to refer to allocated variables. In this process, the suffix ".all" is appended to the name of an access value, delivering the object designated by that access value. For instance, assuming that `Integer_Pointer_1` designates some integer object, the statement

```
Integer_Pointer_1.all := 5;
```

causes that object to take the value 5. As with declared variables, allocated variable references may appear on either side of an assignment statement, or in an expression:

```
Integer_Pointer_1.all := Integer_Pointer_1.all + 1;
```

```
Integer_Pointer_1.all > 4      -- **TRUE**
```

Dynamic allocation involves the evaluation of an expression known as an allocator, which may have one of several forms:

```
new Type or Subtype Name  
new Type or Subtype Name'(Initial Value)  
new Unconstrained Array Subtype Name Index Constraint  
new Unconstrained Record Subtype Name Discriminant Constraint
```

For example, we can declare an access value of the `Color_Pointer_Type` defined above, and provide an allocator:

```
Color_Pointer_1 : Color_Pointer_Type := new Color_Type;
```

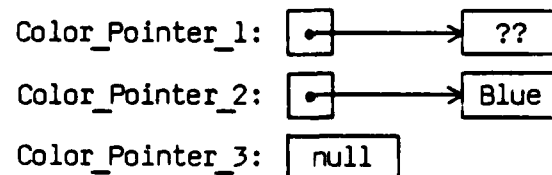
The allocator is of the first and simplest form shown above, with no initial value provided. This statement creates an allocated variable of type `Color_Type`, presumably to be assigned a value later in the program; the value of the expression "new `Color_Type`" is the access value pointing to the new allocated variable, which is assigned to `Color_Pointer_1`. To provide the initial value `Blue` to the object allocated to a different pointer, we could write the following:

```
Color_Pointer_2 : Color_Pointer_Type := new Color_Type'(Blue);
```

As illustrated earlier, an object of an access type need not be initialized. If no allocator is included in the declaration of the object, a null pointer is assigned by default. A null value may also be assigned explicitly:

```
Color_Pointer_3 : Color_Pointer_Type := null;
```

The memory representation of the three access values declared above can be depicted as follows:



The only operations that may take operands of an access type are assignment, equality and inequality. When these are applied, the operands must belong to the same access type; even if `Float_Pointer_Type_1` and `Float_Pointer_Type_2` both define access types designating real numbers, a pointer of `Float_Pointer_Type_1` may not be assigned or compared to a pointer of `Float_Pointer_Type_2`, despite the fact that they both designate values of type `Float`. This situation would appear as follows:

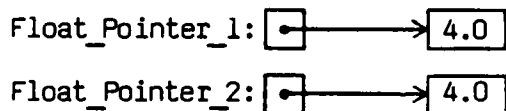
```
type Float_Pointer_Type_1 is access Float;
type Float_Pointer_Type_2 is access Float;
Float_Pointer_1 : Float_Pointer_Type_1 := new Float'(3.0);
Float_Pointer_2 : Float_Pointer_Type_2 := new Float'(4.0);

Float_Pointer_1 := Float_Pointer_2;    -- **ILLEGAL**
```

Note that the objects allocated above are both of type `Float`, so it is legal to write

```
Float_Pointer_1.all := Float_Pointer_2.all
```

which, as in any assignment, copies the right-hand value to the left-hand object. The two pointers still (and always will) refer to different objects, but those objects now hold the same value:



In sum, the collection of objects associated with pointers of one access type may not be shared with (accessed by) pointers of any other access type.





Two access values are equal if and only if they point to the same object. In the declarations below, `Color_Pointer_4` and `Color_Pointer_5` are both initialized to designate objects with the value `Red`; they are not equal,

however, because they refer to distinct variables, as illustrated in the diagram:

```

Color_Pointer_4 : Color_Pointer_Type := new Color_Type'(Red);
Color_Pointer_5 : Color_Pointer_Type := new Color_Type'(Red);

Color_Pointer_4 = Color_Pointer_5      -- **FALSE**
Color_Pointer_4.all = Color_Pointer_5.all -- **TRUE**

Color_Pointer_4:  --> 
Color_Pointer_5:  --> 

```

The allocation of a variable of an unconstrained array type must include either an initial value or an index constraint (forms 2 and 3 of the above list). Observe that an initial value is preceded by an apostrophe, whereas an index constraint is not. Similarly, an initial value or discriminant constraint must be included in the allocation of a variable of an unconstrained record type, unless the discriminant has a default value. Using the unconstrained array type `Bit_Sequence_Type` and corresponding pointer type `Bit_Sequence_Pointer_Type` declared above, we might create the following objects:

```

Bit_Sequence_Pointer_1 : Bit_Sequence_Pointer_Type :=
  new Bit_Sequence_Type'(True, False, True, False);
Bit_Sequence_Pointer_2 : Bit_Sequence_Pointer_Type :=
  new Bit_Sequence_Type (0 .. 3);

```

Both of these pointers now designate 4-element Boolean arrays, but only the first array has been assigned initial values. The dereferencing process may later be used to assign values to the elements of the array designated by `Bit_Sequence_Pointer_2`, or to change the values of the array designated by `Bit_Sequence_Pointer_1`. Again, the suffix `".all"` is appended to the name of the access object to refer to the entire allocated variable, so that

```

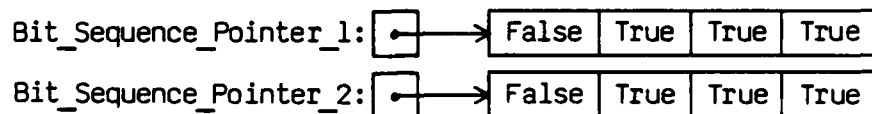
Bit_Sequence_Pointer_1.all

```

indicates the whole array to which `Bit_Sequence_Pointer_1` refers. This expression may be used on either side of an assignment, so one might write the following:

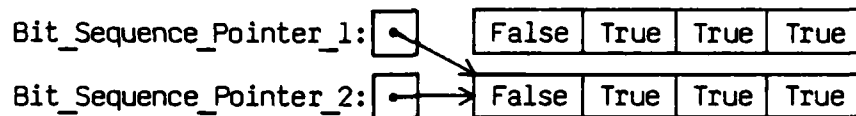
```
Bit_Sequence_Pointer_2.all := (False, True, True, True);
Bit_Sequence_Pointer_1.all := Bit_Sequence_Pointer_2.all;
```

This assignment is legal because the objects specified by ".all" are of the same array type, even though their index bounds may be different; the bounds of the right operand will be converted to those of the left before values are assigned. The above pointers still designate distinct objects, but those objects now hold the same values:



Alternatively, since the two pointers belong to the same access type, they may be assigned to refer to the same object:

```
Bit_Sequence_Pointer_1 := Bit_Sequence_Pointer_2;
```



The above assignment involves the actual access values, not the values of the designated objects. Now an assignment to `Bit_Sequence_Pointer_1.all` will also affect `Bit_Sequence_Pointer_2.all`, and vice versa, which illustrates one of the reasons why access types must be used carefully in order to avoid unexpected changes in data, or "side effects." (Notice that the object formerly designated by `Bit_Sequence_Pointer_1` may still take up space in memory; for all practical purposes, however, it no longer exists, since there is no way to refer to it.)

If an allocated variable belongs to a composite type, the access value may be used to specify particular components of the variable. There are four different types of notation, referring to (1) an entire allocated variable, (2) a component of an allocated array, (3) a slice of an



allocated one-dimensional array, and (4) a component of an allocated record. The first form is shown above, using ".all"; the others are illustrated below:

- |     |                                   |   |                        |
|-----|-----------------------------------|---|------------------------|
| (2) | Bit_Sequence_Pointer_2.all (3)    | \ | Both specify the third |
| OR  | Bit_Sequence_Pointer_2 (3)        | / | array component.       |
| (3) | Bit_Sequence_Pointer_1.all (1..2) | \ | Both specify the       |
| OR  | Bit_Sequence_Pointer_1 (1..2)     | / | array slice.           |
| (4) | Car_Pointer.all.Color_Part        | \ | Both specify the       |
| OR  | Car_Pointer.Color_Part            | / | record component.      |

In form (4), Car\_Pointer is assumed to have been declared as an object of type Car\_Pointer\_Type, as defined above. In all forms, the ".all" is optional.

If an access value designates an array object, the attributes 'First, 'Last, 'Length, and 'Range may be applied to the access value to name the attributes of the designated array. As with component references, the ".all" may be omitted. Using Bit\_Sequence\_Pointer\_2 as declared above, we have:

```
Bit_Sequence_Pointer_2'First = 0
Bit_Sequence_Pointer_2'Last  = 3
Bit_Sequence_Pointer_2'Length = 4
Bit_Sequence_Pointer_2'Range = 0 .. 3
```

A common use of access values occurs in applications that process large units of data, usually records or arrays with many components. Any procedure requiring extensive assignments involving such units becomes inefficient and expensive. In contrast, under typical implementations, access value assignments are fast. Defining and manipulating pointers to the larger units of data thus saves valuable time, requiring only a small amount of extra storage space.

Consider, for example, the following record type:

```
type Huge_Record_Type is
  record
    Component_1      : Long_Integer;
    Component_2      : Some_Long_Array_Type;
    .
    .
    Component_1000   : String (1 .. 100);
  end record;
```

Suppose a program contains 10,000 objects of `Huge_Record_Type`, stored in an array called `Huge_Record_List`, and involves repeated calls to a procedure `Reinsert`. This procedure removes a given record from the array by storing it in a temporary record and then reinserts it into the array at a new position after shifting all records located between the two positions. For example, to move the record at position 500 to position 400, save that record in a local variable, reassign the records from positions 400 to 499 so that they occupy positions 401 to 500 (using slices), and then assign the value of the local variable to record 400:

```
Temporary_Huge_Record := Huge_Record_List (500);
Huge_Record_List (401 .. 500) := Huge_Record_List (400 .. 499);
Huge_Record_List (400) := Temporary_Huge_Record;
```

The `Reinsert` procedure thus contains a slice assignment involving the movement of many huge record values. In the worst case, all 10,000 records would need to be relocated (in moving record 10,000 to position 1). To improve this situation, we instead define an access type:

```
type Huge_Record_Pointer_Type is access Huge_Record_Type;
```

The array components can then be objects of this type. The `Reinsert` procedure would require the same number of assignments in each case, but those assignments would involve access values rather than whole records; by reassigning the pointers to the given records, we logically move the records themselves. When hundreds or thousands of assignments are called for, the access value approach proves to be significantly more efficient.

## PROBLEM

The personnel department of Random, Inc. needs a new filing system to store the large amounts of data associated with each of the company's 1000 employees. The most frequently accessed items of information include full name, social security number, birthdate, and salary, and it would be desirable to have the flexibility to sort the file by any of these fields. (Other information might include starting salary, number of raises, date of next evaluation, title, department, payroll deductions, etc.) Write the type and object declarations to accommodate the required information while promoting efficient sorting operations, keeping in mind that moving large amounts of data is time-consuming and therefore costly. (Don't worry about sorting procedures; they are covered in the Advanced Ada Workbook.)

Write the statements to test whether a given employee makes more than \$40,000 and, if so, to process his/her social security number. These statements should include a call to a procedure Process (declared elsewhere) with the following specification:

```
procedure Process (Social_Security_Number :  
                  in Social_Security_Number_Type);
```

Finally, write the statements to "fire" a given employee (removing all of the corresponding data from the structure) and then fill that position with some overpaid hotshot (represented by the variable Hotshot).

## SOLUTION TO EXERCISE 10.1

### SOLUTION

```

subtype Name_Subtype is String (1 .. 20);
type Month_Type is (January, February, March, April, May, June, July
                    August, September, October, November, December);
type Date_Type is
  record
    Month : Month_Type;
    Day   : Integer range 1 .. 30;
    Year  : Integer range 1900 .. 2100;
  end record;
subtype Social_Security_Number_Subtype is String (1 .. 9);
subtype Salary_Subtype is Integer range 1_000 .. 1_000_000;
type Employee_Count_Type is range 1 .. 1_000;

type Employee_Type is
  record
    First_Name_Part      : Name_Subtype;
    Middle_Initial_Part  : Character;
    Last_Name_Part       : Name_Subtype;
    Social_Security_Number_Part : Social_Security_Number_Subtype;
    Birthdate_Part       : Date_Type;
    Salary_Part          : Salary_Subtype;
    .
    .
    .
  end record;

type Employee_Pointer_Type is access Employee_Type;
type Personnel_File_Type is
  array(Employee_Count_Type) of Employee_Pointer_Type;
Personnel_File : Personnel_File_Type;

To test for salary > $40,000 :

  for I in Employee_Count_Type loop
    if Personnel_File(I).Salary_Part > 40_000 then
      Process (Personnel_File(I).Social_Security_Number_Part);
    end if;
  end loop;

To fire the employee at location X:

  Personnel_File(X) := null;           -- Can no longer access X's data

To hire someone new, filling location X:

  Hotshot : Employee_Type;             -- In declarative part.

begin                                  -- Bring on the replacement
  .
  .
  Personnel_File(X) := new Employee_Type'(Hotshot);
  .
  .
end;
```

## RATIONALE FOR SOLUTION

The structure chosen for the filing system is an array of pointers to employee records. Personnel File belongs to the constrained array type Personnel File Type, indexed by the integer type Employee Count Type; the 1000 components of the array are pointers to objects of the record type Employee Type. These objects hold the relevant data for each employee, and are thus quite large. With this design, sorting operations involve only pointer assignments, avoiding the manipulation of whole records. Personnel File could also be declared as an anonymous array, but then it could not be passed as a parameter, and no other arrays of the same type could be declared. The use of Personnel\_File\_Type avoids these restrictions.

Among the many components of each Employee Type record are those for first and last name, middle initial, social security number, birthdate, and salary. The types Name Subtype (a 20-character string), Date Type (a record type), and Salary Subtype (an integer subtype) are defined and used to specify the types of the corresponding components. The middle initial is a single character, and the social security number is represented by a 9-character string.

To check salaries, the for loop steps through each of the 1000 values in Employee Count, comparing each salary to the value 40,000 and calling Process with the corresponding social security number when appropriate. Individual record components are specified using the notation Access\_Value.Component\_Name, omitting the optional ".all".

To "fire" employee X, the pointer at Personnel\_File(X) is simply assigned the null value. The object previously designated can no longer be accessed, effectively deleting all associated data from the structure. Pointer X is then redirected to the new variable created by the evaluation of the expression "new Employee\_Type." The declared variable Hotshot serves to initialize this new employee record. Equivalently, one could write the following:

```
Personnel_File(X)      := new Employee_Type;  
Personnel_File(X).all := Hotshot;
```

Note that pointer X can not designate Hotshot itself, since the latter is a declared variable and the access value may only refer to allocated variables.

## ALTERNATIVE SOLUTION

```
subtype Name_Subtype is String (1 .. 20);
type Month_Type is . . . -- same as above
type Date_Type is . . . -- same as above
subtype Social_Security_Number_Subtype is String (1 .. 9);
subtype Salary_Subtype is Integer range 1_000 .. 1_000_000;
type Employee_Count_Type is range 1 .. 1_000;

type Employee_Type is
  record
    First_Name_Part      : Name_Subtype;
    Middle_Initial_Part  : Character;
    Last_Name_Part       : Name_Subtype;
    Social_Security_Number_Part : Social_Security_Number_Subtype;
    Birthdate_Part       : Date_Type;
    Salary_Part          : Salary_Subtype;
    .
    .
    .
  end record;

Personnel_File : array (Employee_Count_Type) of Employee_Type;

To test for salary > $40,000 :

  for I in Employee_Count_Type loop
    if Personnel_File(I).Salary_Part > 40_000 then
      Process (Personnel_File(I).Social_Security_Number_Part);
    end if;
  end loop;

To fire the employee at location X:

  Personnel_File(X).First_Name_Part := " ";
  Personnel_File(X).Middle_Initial_Part := ' ';
  Personnel_File(X).Last_Name_Part := " ";
  Personnel_File(X). . .
  ( and so on for each field of the record )

To bring in someone new, filling location X:

  Hotshot : Employee_Type; -- In declarative part
begin -- Bring on the replacement
  . . .
  Personnel_File(X) := Hotshot;
  . . .
end;
```

## DISCUSSION

The alternate solution uses an array of records, saving storage space by eliminating the 1000 access values used in the first solution. The disadvantage to this version is that sorting procedures now involve assignments of whole records, each with possibly hundreds of components. If sorting occurs frequently, the speed of pointer assignment means a significant gain in efficiency.

Record component references are identical in both structures, since the ".all" indicating the object designated by an access value may be omitted when specifying a component. The code to check for salaries over \$40,000 is therefore unchanged in the alternate solution.

In "firing" an employee, however, the first solution has a definite advantage, since a single null access value assignment makes the once-designated record inaccessible, whereas in the alternate solution each component must be reinitialized explicitly. In both versions, the information for a new employee must be assigned component-by-component, here using the variable Hotshot, so neither has a significant advantage in "hiring."

**Section 11**  
**PROGRAM STRUCTURE AND**  
**SEPARATE COMPILATION**

**11.1 Subunits/Library Units**



This page intentionally left blank

## EXERCISE 11.1

### OBJECTIVE

To illustrate subunits and library units.

### TUTORIAL

An Ada program may be compiled in separate pieces. These pieces are library units and secondary units. An advantage in compiling programs, particularly large programs, in separate pieces is that every change made to the program does not require the recompilation of all the parts.

A library unit may be a subprogram declaration, a subprogram body, or a package declaration. They may be compiled separately or grouped together. Once compiled, the unit goes into a program library. Typically, library units are subprogram bodies and package declarations. (Recall that the declaration is the specification followed by a semicolon.) When a library unit is a subprogram body, the subprogram "stands alone" so to speak. It is not enclosed in some other program unit. The "main" procedure, for instance, is a library unit which "stands alone."

Suppose, for instance, that we write a package called Complex Operations that contains several functions and procedures for making calculations with complex numbers:

```

package Complex_Operations is

    type Complex_Type is
        record
            Real: Float;
            Imaginary: Float;
        end record;

    type Complex_Function_Type is array (Integer range<>)
        of Complex_Type;

    function Add_Complex (A, B : Complex_Type) return Complex_Type;
    function Multiply_Complex (A, B : Complex_Type)
        return Complex_Type;
    procedure Graph_Complex_Function (A : in Complex_Function_Type);

end Complex_Operations;

package body Complex_Operations is

    function Add_Complex (A, B : Complex_Type)
        return Complex_Type is
    .
    .
    end Add_Complex;

    function Multiply_Complex (A, B : Complex_Type)
        return Complex_Type is
    .
    .
    end Multiply_Complex;

    procedure Graph_Complex_Function
        (A : in Complex_Function_Type) is
    .
    .
    end Graph_Complex_Function;

end Complex_Operations;

```

The package specification and body could be compiled together.

The functions and procedures in Complex\_Operations may then be used by any program requiring complex calculations. For example, the program Isolate\_Signal could use Complex\_Operations as follows:

```

with Complex_Operations; use Complex_Operations;
procedure Isolate_Signal is
.
.
    F, P, Q, G : Complex_Type;
    Adjusted_Sine_Function : Complex_Function_Type;
begin -- Isolate_Signal
.
.
    F := Add_Complex (P, Q);
    G := Multiply_Complex (P, Q);
.
.
    Graph_Complex_Function (Adjusted_Sine_Function);
.
.
end Isolate_Signal;

```

The "use" statement is not entirely necessary. If it were omitted, the names of the functions and procedures defined in the package would have to be preceded by the name of the package and a period when referred to in the program. For example, the above program could be written:

```

with Complex_Operations;
procedure Isolate_Signal is
.
.
    F, P, Q, G : Complex_Operations.Complex_Type;
    Adjusted_Sine_Function: Complex_Operations.Complex_Function_Type;
begin -- Isolate_Signal
.
.
    F := Complex_Operations.Add_Complex (P, Q);
    G := Complex_Operations.Multiply_Complex (P, Q);
.
.
    Complex_Operations.Graph_Complex_Function (Adjusted_Sine_Function);
.
.
end Isolate_Signal;

```

A "with" clause is placed at the beginning of the subprogram or package specification that it may be discerned, at a glance, that it is required for the succeeding program unit. It may not be placed in an inner scope.

If code that requires several other library units is added to `Isolate_Signal`, for example a package named `Hyperbolic_Trig_Functions` and a procedure named `Find_Echo_Frequency`, there may be several "with" clauses at the beginning of the text,

```
with Complex_Operations;
with Hyperbolic_Trig_Functions;
with Find_Echo_Frequency;
procedure Isolate_Signal is
.
.
end Isolate_Signal;
```

or they may be put in the same "with" clause:

```
with Complex_Operations,
    Hyperbolic_Trig_Functions,
    Find_Echo_Frequency;
procedure Isolate_Signal is
.
.
end Isolate_Signal;
```

Even though `Find_Echo_Frequency` requires the functions in `Machinery_Specs`,

```
with Machinery_Specs;
procedure Find_Echo_Frequency
    (Radar_Reading : in Radar_Reading_Type;
     Echo_Frequency : out Echo_Frequency_Type) is
.
.
end Find_Echo_Frequency;
```

only the name of `Find_Echo_Frequency` need be mentioned for `Isolate_Signal`.

A "with" clause preceding a package or subprogram specification automatically applies to the package or subprogram body. The body is dependent on the specification. A unit using that package is dependent only on the specification.

A unit that is dependent on another unit can only be compiled after the unit on which it depends. If a package specification and body are compiled separately, the specification must be compiled first. If a unit is changed and recompiled, then all of the units which are dependent on it must also be recompiled.

A secondary unit is the body of a package, procedure, or function. Additionally, some secondary units are called subunits, specifically when the unit is a body that has been taken out of another unit. For example, the package above, `Complex_Operations`, could also be written:

```

package Complex_Operations is
.
.
function Add_Complex (A, B : Complex_Type) return Complex_Type;
function Multiply_Complex (A, B : Complex_Type)
    return Complex_Type;
procedure Graph_Complex_Function (A : in Complex_Function_Type);
end Complex_Operations;

package body Complex_Operations is
.
.
function Add_Complex (A, B : Complex_Type)
    return Complex_Type is separate;
function Multiply_Complex (A, B : Complex_Type)
    return Complex_Type is separate;
procedure Graph_Complex_Function (A : in Complex_Function_Type)
    is separate;

end Complex_Operations;

```

The bodies of `Add_Complex`, `Multiply_Complex`, and `Graph_Complex_Function` are replaced by what is called a body stub. The removed units are called subunits. They would be written:

```

separate (Complex_Operations)
function Add_Complex (A, B : Complex_Type) return Complex_Type is
.
.

end Add_Complex;

separate (Complex_Operations)
function Multiply_Complex (A, B : Complex_Type) return Complex_Type is
.
.

end Multiply_Complex;

separate (Complex_Operations)
procedure Graph_Complex_Function (A : in Complex_Function_Type) is
.
.

end Graph_Complex_Function;

```

The word "separate" and the name of the unit from which the body was declared to be separate is written, followed by the body itself.

Visibility within the subunit is exactly as it is at the corresponding body stub. It is exactly as if the procedure or function body existed at the stub. Also, context clauses applying to the parent unit will automatically apply also to the subunit. It should be noted that new context clauses may be introduced when the body of the subunit is defined. These context clauses only have effect for that particular subunit. Thus in the following example,

```
procedure Correlate_Radar_Data is
:
:
function Valid_Track (Track : Track_Type)
return Boolean is separate;

begin -- Correlate_Radar_Data
:
:

end Correlate_Radar_Data;

with Track_Tables;
separate (Correlate_Radar_Data)
function Valid_Track (Track : Track_Type) return Boolean is
:
:

end Valid_Track;
```

Only the function Valid\_Track can access the items declared in Track\_Tables.

In the above examples, the parent unit is a library unit, but it could have been another subunit. If, for instance, Complex\_Operations itself were a body stub taken from a library unit called Numeric\_Operations,

```

package Numeric_Operations is
    :
    package Complex_Operations is
        :
    end Complex_Operations;
end Numeric_Operations;

package body Numeric_Operations is
    :
    package body Complex_Operations is separate;
end Numeric_Operations;

separate (Numeric_Operations)
package body Complex_Operations is
    :
    function Add_Complex (A, B : Complex_Type)
        return Complex_Type is separate;
    function Multiply_Complex (A, B : Complex_Type)
        return Complex_Type is separate;
    procedure Graph_Complex_Function (A : in Complex_Function_Type)
        is separate;

end Complex_Operations;

```

then the body for Add\_Complex, for example, would be written:

```

separate (Numeric_Operations.Complex_Operations)
function Add_Complex (A, B : Complex_Type) return Complex_Type is
    :
end Add_Complex;

```

where dot notation is used to refer to the full name of the package from which the body of Add\_Complex was taken.



One other note - library units must have distinct identifiers within a program library.

Library units and subunits facilitate large systems development in several ways. Library units allow the grouping (or packaging) of related data types, objects (possibly), and operations into single units which can be used throughout a large system. This approach (known as bottom-up) provides the foundation on which a large system can then be built.

The use of subunits allows system development to proceed without premature commitment to implementation decisions. The programmer may assume certain capabilities before actually writing them. In this method of programming (called top-down) the highest level (or structure) of the system can be developed and tested before the rest of the system is finished. The system is completed by successive refinement of the levels. Current software methodology advocates the use of both top-down and bottom-up techniques in large system design and development.

## PROBLEM

A group not familiar with Ada had been given an assignment to develop a utility which allows a user to perform trigonometric operations. Below is their first attempt at a top level design:

```
procedure Trig_Operations (Fn          : in String;
                           Arg         : in Float;   -- in degrees
                           Status_Flag : out Boolean;
                           Result      : out Float) is

  function Sin (Angle : Float) return Float is
  begin -- Sin
    .
  end Sin;

  function Cos (Angle : Float) return Float is
  begin -- Cos
    .
  end Cos;

  function Tan (Angle : Float) return Float is
  begin -- Tan
    .
  end Tan;

begin -- Trig_Operations

  Status_Flag := True;
  if Fn = "Sin" then
    Result := Sin(Arg);
  elsif Fn = "Cos" then
    Result := Cos(Arg);
  elsif Fn = "Tan" then
    Result := Tan(Arg);
  else
    Status_Flag := False;
  end if;

end Trig_Operations;
```

During the preliminary design review, it was decided that the user would want to address the functions directly, rather than by argument to the procedure and that the design could be written in a better Ada style. Rewrite the design to accomplish this.

This page intentionally left blank

## SOLUTION TO EXERCISE 11.1

### SOLUTION

```
package Trig_Operations is

    function Sin (Angle : Float) return Float;           -- in degrees
    function Cos (Angle : Float) return Float;
    function Tan (Angle : Float) return Float;

end Trig_Operations;
```

### RATIONALE FOR SOLUTION

Putting the functions in a package specification enables any user of the package to call these functions directly.

Note that the solution alleviates the need for a Status\_Flag, which tells the user whether the trig operation required is supported by Trig\_Operations, and eliminates the Fn parameter, which tells the procedure which function to call.

### ALTERNATIVE SOLUTION

The following functions compiled as separate library units:

```
function Sin (Angle : Float) return Float is
begin -- Sin
    :
    :
end Sin;

function Cos (Angle : Float) return Float is
begin -- Cos
    :
    :
end Cos;

function Tan (Angle : Float) return Float is
begin -- Tan
    :
    :
end Tan;
```

## DISCUSSION

The alternate solution is perfectly correct. It also allows the user to access each function separately. However, when a system is implemented in this manner, users who need all three functions should need to either write with clauses three times or write one with clause for all three units. For example,

```
with Sin, Cos, Tan;  
procedure Calculations is  
:  
:  
end Calculations;
```

This implementation is slightly more cumbersome, but might have its advantages (ie., when only one of the functions is needed).

However, a possibly greater disadvantage of the alternate solution would be when a decision is made to define all these functions for a type called Radian. The preferred solution, after the change is made, would look like:

```
package Trig_Operations is  
    type Radian is digits 5;  
    function Sin (Angle : Radian) return Float;  
    function Cos (Angle : Radian) return Float;  
    function Tan (Angle : Radian) return Float;  
  
end Trig_Operations;
```

There is no easy way to make this change to the alternate solution. Radian is not a predefined type. A package which defines Radian would have to be defined and this package would have to be "with"ed by all the operations that use that type.

## **Section 12**

### **USING LIBRARY UNITS**

**12.1 Scope and Visibility**

**12.2 Overloading**

**12.3 Private Types**

**12.4 Generics**

This page intentionally left blank

## EXERCISE 12.1

### OBJECTIVE

To introduce scope and visibility.

### TUTORIAL

The scope of a declaration is the region of program text that it potentially influences. It is important to know the exact scope of any entity so that it is properly referred to everywhere in the program, and so that it is not confused with other objects with the same name. Consider, for instance, two loops:

```
First Loop:
for I in 1 .. 10
loop
  .
  .
  for J in 0 .. 1
  loop
    .
    .
  end loop;
end loop;

Second Loop:
for I in Radar'Range
loop
  .
  .
end loop;
```

As explained in Exercise 8.3, the loop parameter has a scope that extends from its declaration in the iterative scheme to the end of that loop. Thus, there can be no confusion between the outer loop parameter in First Loop, I, and the loop parameter in Second Loop. Neither loop parameter is visible outside of its loop. The inner loop parameter of First Loop could be named I because it opens a new declarative region within the declarative region of the outer loop parameter. The inner I would then hide the outer I of First-loop. Within this inner loop, one would have to write First Loop.I in order to refer to the outer I. This practice is strongly discouraged as it detracts from the clarity and maintainability of the code. The use of J as the name of the inner loop parameter here is appropriate and is the recommended programming practice.

The immediate scope of a declaration extends from the declaration to the end of the enclosing declarative region. This enclosing region is called the parent of the declaration. An enclosing region may be: a package declaration or body, a subprogram declaration or body, a record type declaration, a block statement (Block statements will be covered in Exercise 13.1.), or a loop statement.



In the package below,

```
package Signal_Analysis is

    type Signal_Type is (E1, E2, E3, E4, E5);
    Signal_Code : String (1 .. 3);
    procedure Intensify_Signal (A : in Signal_Type;
                                B : out Signal_Type);
end Signal_Analysis;

with Text_IO; use Text_IO;
package body Signal_Analysis is

    procedure Intensify_Signal (A: in Signal_Type; B: out Signal_Type) is
        Voltage : Integer := 500;
        function Isolate_Signal (C : Signal_Type) return Signal_Type is
            Echo_Factor : Integer;
        begin -- Isolate_Signal
            .
            .
        end Isolate_Signal;
    begin -- Intensify_Signal
        .
        .
    end Intensify_Signal;
    .
end Signal_Analysis;
```

the immediate scope of `Signal_Code` extends from its declaration to the end of the package body. The immediate scope of `Voltage` is from its declaration to the end of `Intensify_Signal`, and the immediate scope of `Echo_Factor` is from its declaration to the end of `Isolate_Signal`. If any of these objects was referred to outside of its scope, the compiler would object, calling it an undeclared identifier. Notice that the scope of `Voltage` is only the subprogram body of `Intensify_Signal`. The scope of `Signal_Code` includes most of the package specification of `Signal_Analysis` as well as the entire package body `Signal_Analysis`.

Some declarations have an extended scope which is the full scope of its parent. Declarations with an extended scope are: declarations in the visible part of a package, subprogram formal parameters, and record components.

The immediate scope of a declaration in a package extends from the declaration to the end of the scope of the package. If a package is inside another, the scope of an item declared in the inner package extends to the end of the scope of the outer package. Although these items are in scope, they may or may not be visible. Items are directly visible in their immediate scope. An identifier in an extended scope may be made visible by writing a "use" clause.

In the following package,

```
package P1 is
.
.
package P2 is
    D : Integer;
end P2;
.
end P1;
package body P1 is
.
package body P2 is
.
.
end P2;
.
.
end P1;
```

the object D is in scope from the point at which it is declared until the end of package body P1.

Visibility is normally the same as the scope. There are two types of visibility: direct, where a declared identifier may be used alone to refer to the declared object; and by selection, where the identifier must be preceded by a qualifier, such as a package or record name. For example, record components are directly visible within the record declaration, and visible only by selection outside of the record declaration.

In the following example, the component names of two separate records are the same:

```

procedure Example is

    type A_Rec_Type is
        record
            X : Boolean;
        end record;

    type B_Rec_Type is
        record
            X : Boolean;
        end record;

    A_Rec : A_Rec_Type;
    B_Rec : B_Rec_Type;
    X : Boolean;

begin -- Example
    X := A_Rec.X and B_Rec.X;
    .
    .
end Example;

```

The scope of X in the type A\_Rec\_Type extends from its declaration to the end of the procedure. But the visibility of the component X is confined to the type declaration. Its scope overlaps that of the declared object X. But there is no ambiguity because the regions of visibility do not overlap. The record component is referred to by selection.

Consider the following procedure which uses a block statement:

```

procedure P is

    A : Boolean := True;

begin -- P
    declare

        A : Boolean := False;      -- inner A

    begin

        P.A := A;                  -- outer A gets inner A

    end;

    A := not A;                    -- outer A

end P;

```

The second declaration of A (where it is initialized to False) hides the first A. In other words, after the second declaration of A, a reference to A implies the inner A. If a second A were not declared, a reference to A, even inside the inner scope, would mean the first A, because the scope of A extends to the end of the procedure. After the second declaration of A the first A is said to be hidden, and is referred to by selection, using "P.A". In general, it is better practice to use different names, so that the program is easier to read.

This page intentionally left blank

### PROBLEM

Find what is wrong with the following procedure, and correct it.

```
procedure Initialize_Flight_Plan (A : in Flight_Plan_Type) is
    Fuel_Capacity : Integer := 12000;
    Passengers : Positive;
    .
    .
    procedure Determine_Participant_List (Participants : out Positive) is
        .
        .
        Number_Of_Planes : Positive;
        Plane_Capacity : Positive;

    begin -- Determine_Participant_List
        .
        .
        Participants := Number_Of_Planes * Plane_Capacity;

    end Determine_Participant_List;

begin -- Initialize_Flight_Plan
    .
    .
    Determine_Participant_List (Passengers);
    Requisition_Planes (Determine_Participant_List.Number_Of_Planes);
    .
    .
end Initialize_Flight_Plan;
```

This page intentionally left blank

## SOLUTION TO EXERCISE 12.1

### SOLUTION

The error in the procedure is made where Requisition Planes is invoked. Its argument, the number of planes, is referred to by selection. However, Number\_Of\_Planes is declared in a procedure, and therefore does not have an extended scope. Only declarations in the visible part of a package, subprogram parameters, and record components can be referred to by selection outside of the parent.

The declaration for the number of planes could be moved to the outer procedure:

```
procedure Initialize_Flight_Plan (A : in Flight_Plan_Type) is
    Fuel_Capacity      : Integer := 12000;
    Number_Of_Planes   : Positive;
    Passengers         : Positive;
    .
    .
    procedure Determine_Participant_List (Participants : out Positive); is
        .
        .
        Plane_Capacity : Positive;
    begin -- Determine_Participant_List
        .
        .
        Participants := Number_Of_Planes * Plane_Capacity;
    end Determine_Participant_List;
begin -- Initialize_Flight_Plan
    .
    .
    Determine_Participant_List (Passengers);
    Requisition_Planes (Number_Of_Planes);
    .
    .
end Initialize_Flight_Plan;
```



### RATIONALE FOR SOLUTION

The solution declares Number\_Of\_Planes in Initialize\_Flight\_Plan outside the procedure Determine\_Participant\_List. Number\_Of\_Planes is therefore visible inside the declaration of Determine\_Participant\_List and in the body of Initialize\_Flight\_Plan.

### ALTERNATIVE SOLUTION

```
procedure Initialize_Flight_Plan (A : in Flight_Plan_Type) is
    Fuel_Capacity      : Integer := 12000;
    Number_Of_Planes   : Positive;
    .
    .
    procedure Determine_Participant_List is
        .
        .
        Plane_Capacity   : Positive;
        Planes_To_Be_Used : Positive := Number_Of_Planes;
    begin -- Determine_Participant_List
        .
        .
        Participants := Planes_To_Be_Used * Plane_Capacity;
    end Determine_Participant_List;
begin      -- Initialize_Flight_Plan
    .
    .
    Determine_Participant_List;
    Requisition_Planes (Number_Of_Planes);
    .
    .
end Initialize_Flight_Plan;
```

## DISCUSSION

The alternate solution declares Number\_Of\_Planes in the declarative part of Initialize\_Flight\_Plan and outside the declaration of Determine\_Participant\_List. A separate variable, Planes\_To Be Used, is declared in the declaration of Determine\_Participant\_List, and it is initialized to the value of Number\_Of\_Planes, and used in the calculations of participants.

This solution, while correct, is less desirable because the local variable is unnecessary. No calculations or assignments are done with it. Its only use is as a holding place for the global value.

This page intentionally left blank

## EXERCISE 12.2

### OBJECTIVE

To illustrate overloading.

### TUTORIAL

Within an Ada program under certain circumstances, the same identifier may be used to denote different entities. This is called overloading the identifier. In the following example the identifier Binary is overloaded.

```
type Data_Format_Type is (ASCII, EBCDIC, Binary);  
type Base_Type is (Binary, Hex, Octal, Decimal);
```

The use of Binary in a program which contains both these type declarations, has two possible meanings, the Binary of type Data\_Format\_Type or the Binary of type Base\_Type.

In Ada, identifiers which denote enumeration literals (as seen above), or subprograms, can be overloaded. For example:

```
package Input_Output is  
    procedure Put (X : Integer);  
    procedure Put (X : Complex);  
  
end Input_Output;
```

Here the function Put has been overloaded for Integer and Complex.

In Ada, the operators are considered to be functions. Therefore, overloading may also be defined for them. An Add function for type Complex could be written as,

```
function "+" (X,Y : Complex) return Complex;
```

This is an overloading of the predefined operator "+". For the overloading of operators, the first actual value in the function call corresponds to the first formal parameter and the second actual to the second formal.

Now because the name of the function is an operator, "infix" notation can be used for the function. The adding of two objects A and B of type Complex can be written

A + B

or as

"+"(A,B)

The following operators may be explicitly overloaded.

and	or	xor	<	>	<=	=>
+	-	&	*	/	mod	rem
**	abs	not				

The operators "/" and ":" are not allowed to be explicitly overloaded and the operator "=" is only allowed to be explicitly overloaded for limited private types. (Limited private types are discussed in Exercise 12.3.)

Non-overloadable entities are type and subtype names, object names, labels, block and loop names, package names (including generic packages), task names, exception names, names of entry families, and formal parameters.

A non-overloadable entity and another entity which is named by the same identifier are not allowed to be declared in the same declarative region. For example,

```
package Graphics is
    type Vector is array (Integer range <>) of Float;
    .
    .
    Vector : array (Integer range 1 .. 10) of Float;  -- Illegal
end Graphics;
```

The array object Vector is illegal and would be rejected by the compiler.

When the same identifier is used to name entities that were declared in different declarative regions one entity is said to "hide" the other. For example,

```
package Graphics is
    type Vector is array (Integer range <>) of Float;
    .
    .
    package Chart is
        Vector : array (Integer range 1 .. 10) of Float;
        .
        .
    end Chart;
end Graphics;
```

Inside package Graphics the identifier Vector refers to the object Vector. The type Vector is hidden and can only be referenced by selection.

### PROBLEM

A simulation package needs a facility for specifying the time or scheduling of discrete events. In the system, due to different scheduling applications, time can be represented as Hours, Minutes, or Seconds.

Given the type definitions:

```
package Time_Definitions is
    type Hours is digits 5 range 0.0 .. 100.0;
    type Minutes is digits 5 range 0.0 .. 6000.0;
    type Seconds is digits 5 range 0.0 .. 36000.0;

    end Time_Definitions;
```

Write the specifications for operations to add any time value representation to another time value representation.

(Note that the function bodies need not be implemented. In the function body, one type is converted to the second by whatever conversion is needed. This result is explicitly converted to the desired return type. For example, 1.0 Hour when added to 50.0 Minutes result in 110.0 Minutes. Note also, the capability to add two objects of the same type is the predefined "+" for floating point types.)

This page intentionally left blank

## SOLUTION TO EXERCISE 12.2

### SOLUTION

```
with Time_Definitions; use Time_Definitions;
package Time_Functions is

    function "+" (H: Hours;   M : Minutes) return Hours;
    function "+" (H: Hours;   S : Seconds) return Hours;
    function "+" (M: Minutes; H : Hours)   return Minutes;
    function "+" (M: Minutes; S : Seconds) return Minutes;
    function "+" (S: Seconds; H : Hours)   return Seconds;
    function "+" (S: Seconds; M : Minutes) return Seconds;

end Time_Functions;
```

### RATIONALE FOR SOLUTION

The six overloadings of "+" will enable a use of Time\_Functions to add objects of different time types as if they were of the same type. For example, given the three object declarations below:

```
Seconds_in_Hour    : Seconds := 3600.0;
Minutes_in_Hour    : Minutes := 60.0;
Seconds_in_2_Hours : Seconds;
```

the following computation is now defined.

```
Seconds_in_2_Hours := Seconds_in_Hour + Minutes_in_Hour;
```

The "+" function is the overloaded function which takes a parameter of type Seconds and one of type Minutes and returns a value of type Seconds.



### ALTERNATIVE SOLUTION

```
with Time_Definitions; use Time_Definitions;  
package Time_Functions is
```

```
    function Add_Minutes_To_Hours (H : Hours; M : Minutes)  
        return Hours;  
    function Add_Seconds_To_Hours (H : Hours; S : Seconds)  
        return Hours;  
    function Add_Hours_To_Minutes (M : Minutes; H : Hours)  
        return Minutes;  
    function Add_Seconds_To_Minutes (M : Minutes; S : Seconds)  
        return Minutes;  
    function Add_Hours_To_Seconds (S : Seconds; H : Hours)  
        return Seconds;  
    function Add_Minutes_To_Seconds (S : Seconds; M : Minutes)  
        return Seconds;
```

```
end Time_Functions;
```

### DISCUSSION

The alternate solution is perfectly legal, and provides the user with the same capabilities, ie. to add objects of different time types.

The preference of one solution over the other is highly subjective. It could be argued that the alternate solution is more desirable because the function names describe the functions' actions. In general this is a good argument.

However, when applied in a situation in which an operator symbol exists which portrays the same meaning, the expanded name is hardly preferable. Renaming the predefined "+" operator for Integer to Add\_Integer\_To\_Integer is not any more descriptive of what the function does than leaving the name as "+".

Similarly,.

```
Actual_Wait    : Seconds;  
Minimum_Delay : constant Seconds := 30.0;  
Time_Delay     : Hours;  
.  
.  
Actual_Wait := Minimum_Delay + Time_Delay;
```

is just as clear as writing

```
Actual_Wait := Add_Hours_To_Seconds (Minimum_Delay, Time_Delay);
```

## EXERCISE 12.3

### OBJECTIVE

To introduce private types.

### TUTORIAL

The use of private types allows certain aspects of an implementation to be hidden from the user. In such cases, the programmer cannot access sections of the implementation which he or she uses. The intention of such types is to prevent the misuse of information by extra knowledge of its internal representation, and to prevent, particularly, the programmer's code from becoming dependent upon an implementation and therefore incorrect when the implementation is changed.

For instance, to return to the issue of complex numbers, consider the following package,

```
package Complex_Operations is
  type Complex_Type is
    record
      Real_Part      : Real;
      Imaginary_Part : Real;
    end record;

  function "+" (A, B : Complex_Type) return Complex_Type;
  function "-" (A, B : Complex_Type) return Complex_Type;
  function "*" (A, B : Complex_Type) return Complex_Type;
  function "/" (A, B : Complex_Type) return Complex_Type;

end Complex_Operations;
```

The package above allows the user to see that the type `Complex_Type` is a record. If objects of type `Complex_Type` were created in a program as follows,

```
with Complex_Operations; use Complex_Operations;
procedure Quadratic_Analysis is
  C : Complex_Type;
  D : Complex_Type;
  .
  .
end Quadratic_Analysis;
```

the programmer may, for some reason, be tempted to write

```
C.Imaginary_Part := C.Imaginary_Part + D.Imaginary_Part;  
C.Real_Part := C.Real_Part + D.Real_Part;
```

instead of using the defined complex operation "+", as in

```
C := C + D;
```

If the representation of `Complex_Type` were later changed, perhaps to an array type, the program would be incorrect if the former construct were used.

To avoid these circumstances, the package could be written using private types as follows:

```
package Complex_Operations is  
  
    type Complex_Type is private;  
  
    function "+" (A, B : Complex_Type) return Complex_Type;  
    function "-" (A, B : Complex_Type) return Complex_Type;  
    function "*" (A, B : Complex_Type) return Complex_Type;  
    function "/" (A, B : Complex_Type) return Complex_Type;  
  
private  
    -- Everything after this line is not visible  
    -- to the user of the package.  
    type Complex_Type is  
        record  
            Real_Part      : Real;  
            Imaginary_Part : Real;  
        end record;  
  
end Complex_Operations;
```

The type `Complex_Type` is declared to be private. This means that its representation is not visible to the programmer. In this package, everything before the reserved word "private" is visible, and everything after it is concealed, as noted in the comment.

(Note that the package body for `Complex_Operations` would contain the bodies of the four functions declared.)

Private types may only occur within a package specification. The only operations available outside the package to objects of private types are: assignment (`:=`), equality (`=`), inequality (`/=`), membership (`in`, `not in`), and those provided in the visible part of the package, of which there are, in this case, four. Operations refer to any subprograms declared in the visible (non-private) part of the package which declare parameters of the private type(s).

Limited private types are a special form of private types which permit only those operations specified in the visible part of the package. The advantage of using limited private types is that the writer of the package has even greater control over how the declared types are used outside the package.

The format for package specifications using private and limited private types is:

```
package Package_Name is

    type Type_Name_1 is private;
    type Type_Name_2 is limited private;
    .
private
    type Type_Name_1 is < Actual Representation > ;
    type Type_Name_2 is < Actual Representation > ;
    .
end Package_Name;
```

Private and limited private types can be used for any objects. Their only difference is the restriction of available operations.

AD-A165 345

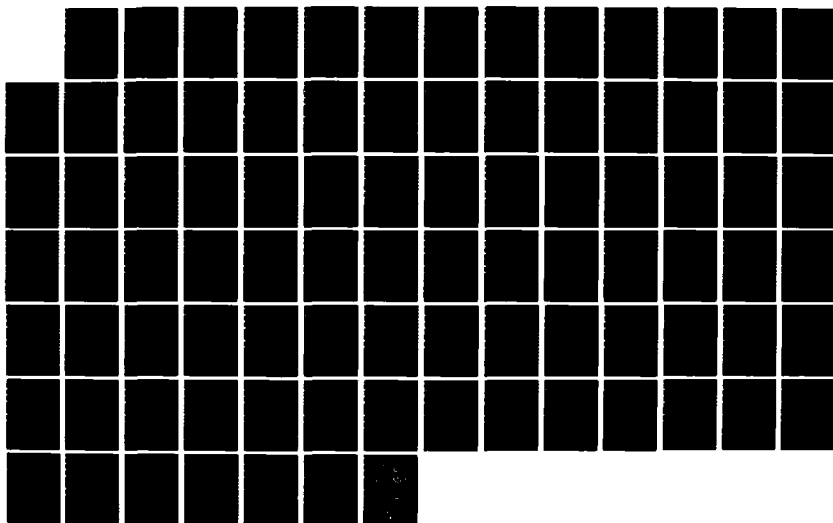
ADA (TRADEMARK) PRIMER(U) SOFTECH INC WALTHAM MA 1986  
DAA807-83-C-K586

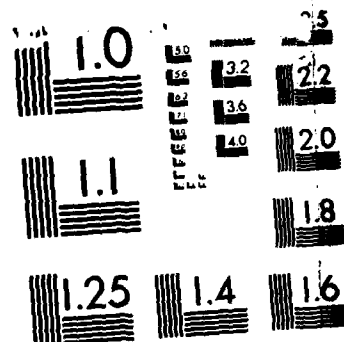
5/8

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A

This page intentionally left blank

PROBLEM

Write a package specification called Stack\_Pkg that contains the types and procedures necessary to declare a stack object, and subsequently push or pop integers onto or off of it.



This page intentionally left blank

## SOLUTION TO EXERCISE 12.3

### SOLUTION

```
package Stack_Pkg is

  type Stack_Type is private;

  procedure Push (A : in out Stack_Type; X : in Integer);
  procedure Pop  (A : in out Stack_Type; X : out Integer);

  -- Exceptional case like popping an empty stack or pushing onto
  -- a full stack are not handled.

private

  Max_Stack_Length : constant Integer := 100;
  type Arr_Type is array (1 .. Max_Stack_Length) of Integer;
  type Stack_Type is
    record
      Contents : Arr_Type;
      Top      : Integer range 0 .. Max_Stack_Length := 0;
    end record;

end Stack_Pkg;
```

### RATIONALE FOR SOLUTION

Stack\_Pkg contains the definition of the type for the Stack, Stack\_Type, its maximum length, Max Stack\_Length, and two procedures, Push and Pop. The types of the stack and of its length are declared private. Stack\_Type is a record with two components: Contents, an array of length Max Stack\_Length for the integers in the stack, and Top, which stores the index in the array of the current top of the stack.

Stack is declared to be private so that the user cannot know exactly how it is represented, and thereby tamper with it.

### ALTERNATIVE SOLUTION

```
package Stack_Pkg is
    Max_Stack_Length : constant Integer := 100;

    type Stack_Type is
        record
            Contents : array (1 .. Max_Stack_Length) of Integer;
            Top      : Integer range 0 .. Max_Stack_Length := 0;
        end record;

    procedure Push (A : in out Stack_Type; X : in Integer);
    procedure Pop  (A : in out Stack_Type; X : out Integer);

end;
```

### DISCUSSION

The alternate solution is the same as the first solution except that it does not use private types. The user of this package can see how the stack is represented (and how long it is) and potentially misuse the capability. But more importantly, if the representation of Stack\_Type or its length Max\_Stack\_Length were changed (which is a valid possibility) a program using these constructs may become incorrect.

## EXERCISE 12.4

### OBJECTIVE

To introduce generic program units.

### TUTORIAL

Consider, again, a package specification for a stack:

```
package Int_Stack is
    type Stack_Type is private;
    procedure Push (X : in Integer);
    function Pop return Integer;

private
    Max_Stack_Length : constant Integer := 100;
    Stack_Type is array (1 .. Max_Stack_Length) of Integer;
end Int_Stack;
```

A program may require several different stacks for different types, say real or enumeration, and of different lengths. Instead of writing a separate package for each stack, one can write a single generic package.

A generic mechanism allows packages and subprograms to have parameters that may be types, objects, or subprograms. Their use eliminates the repetition of similar code. In the case of the package specification for a stack, we could write a single generic package that takes two parameters, a type that indicates the type of the elements of the stack, and a natural number that determines the length of the stack. The package would look like this:

```
generic
  Max_Stack_Length : Natural;
  type Item_Type is private;
package Stack is

  procedure Push (X : in Item_Type);
  function Pop return Item_Type;

end Stack;
```

We can now create several stacks from this generic template as follows:

```
type Real is digits 5;
package Int_Stack is new Stack (100, Integer);
package Real_Stack is new Stack (50, Real);
subtype Small_Integer_Type is Integer range 1 .. 15;
package Small_Int_Stack is new Stack (100, Small_Integer_Type);
```

These declarations are called instantiations. They create actual code from a generic template. The format for an instantiation is:

```
package Package_Name is new Generic_Package_Name (Parameter_List);
```

The formats for instantiations of generic subprograms are:

```
procedure Procedure_Name is new Generic_Procedure_Name (Parameter_List);
function Function_Name is new Generic_Function_Name (Parameter_List);
```

A simple example of a generic function is one that finds the next element of a discrete type, i.e. integer or enumeration, and returns the first element when called with the last.

```
generic
  type Item_Type is ( < > );
  function Next (X : Item_Type) return Item_Type;

  function Next (X : Item_Type) return Item_Type is
  begin -- Next

    if X = Item_Type'Last then
      return Item_Type'First;
    else
      return Item_Type'Succ(X);
    end if;

  end Next;
```

Item\_Type is the formal parameter. It stipulates that the actual parameter must be a discrete type. To instantiate Next with an enumeration type,

```
type Exercise_Name is
  (Big_Pine_II, Unit, Universal_Trek, Readex,
   Kindle_Liberty, Big_Pine_I, Ocean_Venture);
```

we would write

```
function Next_Exercise is new Next (Exercise_Name);
```

Notice that the type parameter for a discrete type in the function Next is written

```
type T is ( < > );
```

The possible generic type parameters are written:

```
type T is private;           -- private type
type T is limited private;   -- limited private type
type T is ( < > );           -- discrete type
type T is range < > ;        -- integer type
type T is digits < > ;        -- floating point type
type T is delta < > ;         -- fixed point type
```

Also, array type declarations are allowed as generic parameters, as demonstrated in the exercise.

This page intentionally left blank

PROBLEM

Write a generic function that will sum the elements of a one-dimensional array having any integer component type and any index type.



This page intentionally left blank

## SOLUTION TO EXERCISE 12.4

### SOLUTION

```
generic
  type Index_Type is (<>);
  type Integer_Type is range <>;
  type Array_Type is (Index_Type range <>) of Integer_Type;
function Sum (A : Array_Type) return Integer_Type;

function Sum (A : Array_Type) return Integer_Type is
  Result : Integer_Type := 0;
begin -- Sum
  for I in A'Range loop
    Result := Result + A(I);
  end loop;
  return Result;

end Sum;
```

### RATIONALE FOR SOLUTION

Sum is a generic function having three parameters, Index\_Type, the type of the index of the array, Integer\_Type, the type of the component of the array, and Array\_Type the type of the array itself. Each is declared with the standard format for generic parameters of those types. The function itself declares an object Result of the same type as the components of the array, and initializes it to zero. Using a for loop with an index range A'Range each component is successively added to Result, and then Result is returned.

Without declaring a generic function, this problem would require indefinite numbers of separate function declarations, one for each type of array used.

### ALTERNATIVE SOLUTION

There is no alternate solution which meets the stipulations of the problem.

A change to the specification of Index\_Type to "range <>" or "private" would change its meaning. In the first case, "range <>" would eliminate enumeration types from being the index type. Using "private" here would forbid the use of the type as an index type because the index type must be discrete.

Likewise, a change in the specification of Integer\_Type to "<>" or "private" would again drastically change the meaning of the array type. Specifically "+" would not necessarily be available for the possible type that would be used if the changes were incorporated. The component type would no longer be restricted only to integer types.

#### DISCUSSION

Generic units allow more effective reuse of common code. A package or operation that performs the same function for different types can be written once, and instantiated many times. The program is often more readable and closer to the problem in representation of structure and function.

## **Section 13**

### **PACKAGES**

#### **13.1 Interface vs. Implementation**

This page intentionally left blank

## EXERCISE 13.1

### OBJECTIVE

To expand on earlier discussions of packages, and to demonstrate uses of context clauses and renaming declarations in package specifications and bodies.

### TUTORIAL

The concept of an Ada package was introduced in Exercise 2.1 as a mechanism used to group together logically related entities, such as variables, constants, type declarations, subprograms, tasks, or even other packages. A package provides a collection of computational resources that might be used by other program units, serving as a building block for larger programs. For instance, the package Simple Graphics presented in Exercise 2.1 provided several basic graphics procedures that were used by the procedure Draw Flag in order to draw a flag. A widely used example is the Text\_IO package described in Section 15, which supplies all facilities needed for input and output of text.

Packages give Ada the ability to fulfill several programming goals. They facilitate modularity, breaking down huge programs into manageable units; they allow the division of a large project among independent programming teams; and they promote the creation of general-purpose software libraries. The package structure of an Ada program is often the key to its design and maintainability.

Related to maintenance and modularity is the principle of "separation of concerns." Because of the distinction between the interface, or external appearance of a package, and its implementation, or internal organization and algorithms, it is not necessary to consider the implementation and the use of the package at the same time. The package specification provides the user of the package with all information necessary to use its resources, while withholding the details of how those resources work internally. The maintenance programmer is then able to make implementation changes without affecting how the package is used.

The interface of a package is also known as its visible part, since it contains information that is to be "seen" by, and thus available to, the package user. This visible information may include (among other things) variable declarations, type definitions, and subprogram specifications, with all parameter names and types, and result types for functions. Such declarative items appear at the beginning of the package specification; they may be followed by a private section, containing definitions for any types declared private or limited private, plus declarations of any constants of those types (called deferred constants). The Calendar package discussed below provides an example of a specification with both visible and private sections. Although names of private types are

visible to the user, definitions found in the private part are not available; they are thus part of the implementation rather than the interface.

The rest of the implementation details are found in the body of the package. Bodies (or body stubs) for all subprograms declared in the specification must appear in the package body; in addition, the body may include internal variables, types, and subprograms, all of which are hidden from the user and may not be referenced from outside the package. Context clauses may also be used to import resources directly to the package body, contributing to the implementation without complicating the interface.

The structure of a package is as follows, with optional sections denoted by brackets:

Specification:

```
package Package_Name is
    Sequence of Declarations;
    [private
        Sequence of Declarations;]
end [Package_Name];
```

Body:

```
package body Package_Name is
    Sequence of Declarations;
    [begin -- Package_Name
        Sequence of Initialization Statements;]
end [Package_Name];
```

In a package body, the word begin and the statements following it are optional. These statements are executed once, when the package comes into existence, and are generally used to initialize variables used by the package. For example, a package might contain a look-up table of conversion values that must be read from a file before the package can be used; the body would then contain statements to open the file, read the data into the table and close the file again. Similarly, in the Factorial\_Table\_Manager package below, the procedure Initialize\_Factorial\_Table is used to fill the array Factorial\_Table:

```
package Factorial_Table_Manager is
  subtype Table_Index_Subtype is Integer range 1 .. 1000;
  Table_Pointer : Table_Index_Subtype;
  Factorial_Table : array (Table_Index_Subtype) of Integer;
  procedure Initialize_Factorial_Table;
end Factorial_Table_Manager;

package body Factorial_Table_Manager is

  function Factorial_Of (N : Integer) return Integer is
    Fact : Integer := 1;
  begin
    for I in 1 .. N loop      -- Compute factorial of N
      Fact := Fact * I;
    end loop;
    return Fact;
  end Factorial_Of;

  procedure Initialize_Factorial_Table is
  begin
    for N in Factorial_Table'Range loop
      Factorial_Table (N) := Factorial_Of (N);
    end loop;
  end Initialize_Factorial_Table;
begin
  Initialize_Factorial_Table;
end Factorial_Table_Manager;
```

Since the specification for Initialize Factorial Table is visible to the user, this routine may be called by other units Importing Factorial\_Table\_Manager. Alternatively, the procedure specification could be removed from the visible part of the package, making Initialize Factorial Table local to the package body, as is the Factorial\_Of function. Another option would be to move the procedure body statements to the executable part of the package body, removing the procedure call and thereby eliminating the need for a procedure. With



either alternative, the user no longer has access to the initialization routine. If `Initialize_Factorial_Table` were removed, `Factorial_Table_Manager` would appear as follows:

```
package Factorial_Table_Manager is
  subtype Table_Index_Subtype is Integer range 1 .. 1000;
  Table_Pointer : Table_Index_Subtype;
  Factorial_Table : array (Table_Index_Subtype) of Integer;
end Factorial_Table_Manager;

package body Factorial_Table_Manager is

  function Factorial_Of (N : Integer) return Integer is
    Fact : Integer := 1;
  begin
    for I in 1 .. N loop
      Fact := Fact * I;
    end loop;
    return Fact;
  end Factorial_Of;

begin
  for N in Factorial_Table'Range loop
    Factorial_Table(N) := Factorial_Of (N);
  end loop;
end Factorial_Table_Manager;
```

An entity declared in a package specification is directly visible and may be referred to by its identifier anywhere in the specification after its declaration and anywhere in the package body. In the `Factorial_Table_Manager` package above, the type `Table_Index_Subtype` is declared in the specification and used later in that section to provide a range for the index of `Factorial_Table` and a type for `Table_Pointer`. The array `Factorial_Table`, also declared in the specification, is assigned values by statements in the body.

Outside the package, a visible entity may be referred to by its expanded name, which takes the form:

Package\_Name.Identifier

Thus a unit mentioning `Factorial_Table_Manager` in a `with` clause could refer to `Factorial_Table_Manager.Factorial_Table` or call `Factorial_Table_Manager.Initialize_Factorial_Table` (if the procedure is left as first shown).

As noted above, a package body may include its own declarations, creating entities that are not part of the package interface. A variable declared in a package body exists for the lifetime of the package; if one of the package's subprograms (or initialization statements) leaves a value in such a variable, that value will be there the next time one of the subprograms is called. The same holds for variables declared in the private part of the package. An example might be a package that keeps a running count of the number of changes made to a file via certain procedure calls, where the physical object containing the count is not accessible to the user but a function is provided to return the value.

Package bodies may also contain subprograms that are not included in the package specification, like the `Factorial_Of` function in the body of `Factorial_Table_Manager` above. Such subprograms may only be called from within the package body, either by other subprograms or by statements in the executable part of the package. Any procedure or function that is not to be available to the user should be declared in the body, as part of the package's implementation.

Note that every package must have a specification, but not all packages require a body. A body must be provided if the package specification contains at least one declaration of a unit that needs a body; examples of such units are subprograms, generic units, and (possibly) packages. The second version of `Table_Manager` shown above does not require a body but includes one to carry out the initialization routine. The package below contains only type definitions and object declarations, so it needs no body:

```
package Work_Data_Package is
  type Day_Type is (Monday, Tuesday, Wednesday, Thursday,
                    Friday, Saturday, Sunday);
  subtype Work_Day_Type is Day_Type range Monday .. Friday;
  type Hours_Worked is delta 0.25 range 0.0 .. 24.0;
  type Time_Table_Type is array (Day_Type) of Hours_Worked;

  Normal_Hours : constant Time_Table_Type :=
    (Work_Day_Type => 8.0,
     Saturday | Sunday => 0.0);
end Work_Data_Package;
```

`Work_Data_Package` describes a common pool of objects and types. If there is a need for subprograms to operate on the work data, those subprograms should also be included in the package (creating the need for a package body). Occasionally, a package might contain only variable and constant declarations, creating a global data pool; in general, however, packages are best utilized when they group type and object declarations with any associated subprograms.

The packages used to build programs may be user-defined, like Work Data Package, or they may be predefined, like package Text IO described in section 15. Another predefined package available to all Ada programs is called Calendar; it provides a good example of the interface/implementation distinction in that it declares a private type Time whose actual representation may vary from one implementation to the next. The specification of Calendar is as follows:

```

package Calendar is

  type Time is private;

  subtype Year_Number   is Integer   range 1901 .. 2099;
  subtype Month_Number  is Integer   range 1 .. 12;
  subtype Day_Number    is Integer   range 1 .. 31;
  subtype Day_Duration  is Duration range 0.0 .. 86_400.0;

  function Clock return Time;

  function Year   (Date : Time) return Year_Number;
  function Month  (Date : Time) return Month_Number;
  function Day    (Date : Time) return Day_Number;
  function Seconds (Date : Time) return Day_Duration;

  procedure Split (Date      : in Time;
                  Year       : out Year_Number;
                  Month      : out Month_Number;
                  Day        : out Day_Number;
                  Seconds    : out Day_Duration);

  function Time_Of (Year      : Year_Number;
                  Month      : Month_Number;
                  Day        : Day_Number;
                  Seconds    : Day_Duration := 0.0) return Time;

  function "+" (Left : Time;      Right : Duration) return Time;
  function "+" (Left : Duration; Right : Time)      return Time;
  function "-" (Left : Time;      Right : Duration) return Time;
  function "-" (Left : Time;      Right : Time)      return Duration;

  function "<" (Left, Right : Time) return Boolean;
  function "<=" (Left, Right : Time) return Boolean;
  function ">" (Left, Right : Time) return Boolean;
  function ">=" (Left, Right : Time) return Boolean;

  Time_Error : exception; -- can be raised by Time_Of, "+", "-"

private
  -- implementation dependent
end;
```

Calendar uses the predefined fixed point type Duration, expressing values in seconds; the subtype Day\_Duration ranges from 0 to 86400, the number of seconds in a day. The parameterless Clock function returns the current value of Time at the time it is called, so the following line might appear in a program importing Calendar:

```
Current_Time := Clock;
```

Here it is assumed that Current\_Time is a variable of type Time and that Calendar appears in both a with and a use clause, making Clock directly visible. The value now in Current\_Time is the time at which Clock was evaluated.

Also in Calendar are the functions Year, Month, Day, and Seconds, which return the corresponding values for a given value of type Time, and the procedure Split that returns all four values. Conversely, the function Time Of combines a year, month, day, and duration into a Time value. In addition, Calendar includes overloaded addition and relational operators, and the exception Time\_Error, which is raised by Time Of if the parameters do not form a proper date or by "+" or "-" if the result is out of range (see section 14 on exceptions). The Problem part of this exercise presents one situation in which some of Calendar's resources might be utilized.

Once a package (or any library unit) has been compiled, it may be named in the context clause of another program unit (see exercise 11.1 for more discussion of context clauses). A with clause naming a package makes the entities in the package interface available to the program by means of expanded names, as shown above. If the package is also mentioned in a use clause, the entities in its specification may be referred to by simple names (identifiers) rather than by expanded names. A use clause thus changes only the way references to entities may be made; it has no effect on the actual accessibility of the entities involved.

Although use clauses simplify references to imported entities by eliminating long expanded names, they sometimes make it difficult to determine where an entity came from, especially if a program mentions several units in its context clauses. A use clause is appropriate when the entities being accessed are familiar, such as the Get and Put procedures of Text\_IO, or when new versions of operators are provided, as in Calendar. In the latter case, the use clause allows the operator to appear between the operands, as it would normally, rather than as a

function call. For example, a program naming Calendar in a use clause could declare variables Time\_In and Time\_Out of type Time and later subtract the values of those variables using the expression

Time\_Out - Time\_In

rather than

Calendar."-" (Time\_Out, Time\_In)

Renaming declarations provide an alternative to use clauses, allowing an imported entity to take on a more succinct or meaningful name while documenting that entity's origin. These local declarations describe each entity completely and thus require more writing than do use clauses, but they generally make programs much easier to understand. Using a with clause alone, we always know where every entity comes from, but adding a use clause allows us to write short names; renaming helps achieve a balance between the two situations.

Objects, exceptions, subprograms, and packages may be renamed; in addition, a subtype declaration may be used to achieve the effect of renaming a type or subtype. Examples are illustrated below:

```
subtype Time_Sheet_Type is Work_Data_Package.Time_Table_Type;
Standard_Time_Sheet : constant Time_Sheet_Type
                      renames Work_Data_Package.Normal_Hours;
```

```
Time_Error : exception renames Calendar.Time_Error;
```

```
subtype Absolute_Time is Calendar.Time;
function "<"(Left, Right: Time) return Boolean
                      renames Calendar."<";
```

```
package Some_Package renames Imported_Package.Embedded_Package;
```

Note that the new name may be identical to all or part of the old name, but it need not be.

## PROBLEM

Suppose a simple message-handling program needs some way to log all messages that come in during program execution. This logging procedure will involve storing each message with the time at which it was received and the elapsed time since the last message came in (for later statistical analysis). The program must also be able to retrieve the oldest message currently in the log (so that the next-oldest becomes the current oldest), and it should be able to find out the total number of messages logged. Write a package called `Message_Log_Package` containing the types, objects, and subprograms necessary to fill these needs.

The specification of `Message_Log_Package` should provide the user with all the information needed to carry out the operations described above. The definition of `Message_Type` should appear here, since the program importing the package must be able to declare the message objects that will be passed to the logging procedure. To allow messages of any length, use a variable-length string type, as illustrated in Exercise 9.4 on record types with discriminants. Three subprograms must also be available to the user, so specifications for a `Log_Message` procedure, a `Retrieve_Oldest_Message` function, and a `Total_Messages_Logged` function should appear in the visible part of the package, with appropriate parameters and return values. Subprograms to manipulate and return time readings need not be included at this point; assume that they might be added in the future.

The facilities for handling time values should come from the `Calendar` package shown above; think carefully about where and how `Calendar`'s resources should be imported. For instance, a variable of type `Time` can keep track of each logging time, so that at the next call to `Log_Message` it can be used to compute the length of the interval since the last message was logged. This variable should be initialized to the time the package is created, using the `Clock` function.

There are several ways that `Message_Log_Package` might be implemented. A simple way to represent the log is as an array of records, with one record holding the information for a particular message. The array can then be filled in order, with the first message stored in the first position; two "pointer" variables keep track of the positions of the next entry to be retrieved and the next location to be filled. A third variable counts the total number of messages logged; this value will be returned by the `Total_Messages_Logged` function, keeping the actual object from being directly accessible by the user.

This solution requires that a maximum number of messages be defined, to provide an upper bound for the array index. The log should be able to accommodate up to 250 messages for now; this number could change in the future. Error conditions, such as running out of room in the array or trying to retrieve a message after all have been retrieved, would generally be handled by exceptions, as discussed in Exercise 13.1. For

the purposes of this exercise, however, overflow may be dealt with as follows: at each call to `Log_Message`, check the value of the variable indicating the next entry location; when it holds the last array index value (now 250), reset it to the first value and continue to enter new messages. Similarly, the variable holding the next retrieval location would be checked on calls to `Retrieve_Oldest_Message`, and reset if necessary. This "wrap around" method could result in the loss of information if the oldest messages have not yet been retrieved before the array becomes full; a solution using exceptions would be preferable. When the last message has been retrieved, the next call to the retrieval function might simply return a blank (null) message, although again an exception would normally be used to report the situation. Alternate solutions using access types would eliminate the possibility of such overflow; these methods are presented in the Advanced Ada Workbook.

The `Log_Message` procedure should install a message object, passed as a parameter, at the position indicated by the variable holding the next entry location. It will also need to obtain and store the current time and compute the time since last logging, so the package must provide the capability to find the difference between two time values, with a result of type `Duration`. Finally, this procedure must increment the entry-location variable as well as the variable counting the total messages logged.

`Retrieve_Next_Message` returns the current "oldest" message, obtaining the correct position from the appropriate "pointer" variable. You need not write the actual statements implementing this function or those for `Total_Messages_Logged`; instead, indicate by dots or by an "is separate" clause where those bodies would normally be found. (The function bodies will be included in the Solution for completeness.)

Other declarations may be needed to complete the package as outlined. Think about which entities should be visible to the main program, and which should be hidden in the implementation. Remember that the implementation chosen here is not necessarily the most efficient; careful package design will allow improvements to be made to the body without changing what is available to the user.

## SOLUTION TO EXERCISE 13.1

### SOLUTION

```
package Message_Log_Package is

  type Message_Type (Length : Natural := 0) is
    record
      Text_Part : String (1 .. Length);
    end record;

  procedure Log_Message (Incoming_Message : in Message_Type);

  function Retrieve_Oldest_Message return Message_Type;
  function Total_Messages_Logged return Natural;

end Message_Log_Package;

with Calendar;
package body Message_Log_Package is

  subtype Absolute_Time is Calendar.Time;
  Last_Time_Logged : Absolute_Time;

  type Message_Log_Entry_Type is
    record
      Message_Part      : Message_Type;
      Time_Received     : Absolute_Time;
      Time_Since_Last   : Duration;
    end record;

  subtype Message_Log_Index_Subtype is Integer range 1 .. 250;

  Message_Log : array (Message_Log_Index_Subtype)
    of Message_Log_Entry_Type;

  Total_Received : Natural := 0;
  Next_Retrieval_Location : Message_Log_Index_Subtype := 1;
  Next_Entry_Location : Message_Log_Index_Subtype := 1;

  function Current_Time return Absolute_Time renames Calendar.Clock;
  function "-" (Left, Right : Absolute_Time) return Duration
    renames Calendar."-";

  (Continued on next page)
```



```

procedure Log_Message (Incoming_Message : in Message_Type) is
    Interval : Duration;
    Local_Message_Entry : Message_Log_Entry_Type renames
        Message_Log (Next_Entry_Location);
begin
    -- Fill log entry:
    Local_Message_Entry.Message_Part := Incoming_Message;
    Local_Message_Entry.Time_Received := Current_Time;
    Interval := Local_Message_Entry.Time_Received - Last_Time_Logged;
    Local_Message_Entry.Time_Since_Last := Interval;

    -- Update global variables:
    Last_Time_Logged := Local_Message_Entry.Time_Received;
    Total_Received := Total_Received + 1;

    if Next_Entry_Location < Message_Log_Index_Subtype'Last then
        Next_Entry_Location := Next_Entry_Location + 1;
    else
        Next_Entry_Location := Message_Log_Index_Subtype'First;
    end if;
end Log_Message;

function Retrieve_Oldest_Message return Message_Type is
    Return_Location : Message_Log_Index_Subtype :=
        Next_Retrieval_Location;
begin
    if Next_Retrieval_Location < Message_Log_Index_Subtype'Last then
        Next_Retrieval_Location := Next_Retrieval_Location + 1;
    else
        Next_Retrieval_Location := Message_Log_Index_Subtype'First;
    end if;
    return Message_Log (Return_Location).Message_Part;
end Retrieve_Oldest_Message;

function Total_Messages_Logged return Natural is
begin
    return Total_Received;
end Total_Messages_Logged;

begin
    Last_Time_Logged := Current_Time;

end Message_Log_Package;

```

## RATIONALE FOR SOLUTION

The specification of `Message_Log_Package` contains only the `Message_Type` definition and the specifications of the routines for logging messages, retrieving the oldest message, and delivering the number of messages logged. These are the only resources needed by the package user; all other declarations are hidden in the body, creating a simple, concise package interface.

The messages stored in the log will belong to the visible type `Message_Type`, defined as a record with one discriminant, `Length`; the discriminant is given a default initial value of zero so that unconstrained message objects may be declared, as suggested in Exercise 9.4. The other component is a string whose length is determined by the value of the discriminant; it will hold the actual text of the message. The specification for the procedure `Log_Message` shows that it takes in an object of type `Message_Type`; since this parameter is of mode `in`, no values are returned by the routine. Both functions declared are parameterless: `Retrieve_Oldest_Message` returns a value of type `Message_Type`, and `Total_Messages_Logged` returns a natural number (zero or greater).

The package body is preceded by a `with` clause naming the library package `Calendar`. All entities declared in the specification of `Calendar` (presented in the Tutorial) are thus available for use in the body of `Message_Log_Package` and may be referred to by their expanded names. Since `Calendar`'s resources are not needed in the specification, the context clause is associated only with the package body, further separating the implementation details from the interface.

Because `Calendar` is not mentioned in a `use` clause, expanded names must be used in references to the entities it imports. Renaming declarations are utilized here both to document which objects, types, and subprograms come from `Calendar`, and to provide shorter or more descriptive names for such entities. The declaration of the subtype `Absolute_Time` in effect renames `Calendar`'s private type `Time`; that subtype then serves as the type for the object `Last_Time_Logged`. Also imported from `Calendar` are the `Clock` function, renamed `Current_Time`, and the overloaded subtraction operator that finds the difference between two values of type `Time` (or `Absolute_Time`, as renamed here). The local renaming of `"-"` allows it to be used as an infix (between the operands) rather than as a prefix, as discussed in the Tutorial.

The body of Message\_Log\_Package also contains several local types and objects. Message\_Log\_Entry\_Type is a record type defining the structure of each log entry; it provides components to hold the message itself, the time received, and the time since the last message was received. The log itself is represented by the array Message\_Log, declared as an object of an anonymous array type. The array components are of the entry type described above, and its index bounds are defined by the Integer subtype Message\_Log\_Index\_Subtype, which ranges from 1 to 250. Use of the subtype to provide the index range allows for easier modification, since the size of the log could change. Furthermore, by declaring the variables Next\_Retrieval\_Location and Next\_Entry\_Location to be of this subtype, we automatically ensure that those variables may only take on values within the range of the array. Finally, Total\_Received is declared as an object of type Natural and is initialized to zero, since no messages have yet been logged.

As required, the body of Message\_Log\_Package contains the bodies of the three subprograms declared in the specification. Log\_Message declares a local variable Interval of type Duration, used for readability in computing the time since last logging. It also illustrates another use of renaming declarations, in declaring Local\_Message\_Entry to provide a shorter name for Message\_Log (Next\_Entry\_Location). The body of Log\_Message includes statements to fill in the appropriate log entry and to update Last\_Time\_Logged, Total\_Received, and Next\_Entry\_Location, checking for overflow as specified. The Time\_Received component takes the value returned by a call to Current\_Time; by subtracting Last\_Time\_Logged from this value, we obtain the Duration value for Time\_Since\_Last.

The two function bodies are straightforward, serving mainly to deliver values hidden by the implementation. The message retrieval function increments Next\_Retrieval\_Location, again adjusting for possible overflow, and returns the message indicated by the old value of that variable, held in the local variable Return\_Location. Total\_Messages\_Logged simply returns the current value of Total\_Received, preventing that value from being altered by the user.

Message\_Log\_Package also has an executable section used to initialize Last\_Time\_Logged with a call to Current\_Time. The logging process is thus considered to begin when the package comes into existence.

### ALTERNATIVE SOLUTION

```
with Calendar; use Calendar;
package Message_Log_Package is

  type Message_Type (Length : Natural := 0) is
    record
      Text_Part : String (1 .. Length);
    end record;

  type Message_Log_Entry_Type is
    record
      Message_Part      : Message_Type;
      Time_Received     : Time;
      Time_Since_Last   : Duration;
    end record;

  subtype Message_Log_Index_Subtype is Integer range 1 .. 250;

  Message_Log : array (Message_Log_Index_Subtype)
    of Message_Log_Entry_Type;

  Next_Retrieval_Location : Message_Log_Index_Subtype := 1;
  Next_Entry_Location     : Message_Log_Index_Subtype := 1;
  Last_Time_Logged        : Time := Clock;

  procedure Log_Message (Incoming_Message : in Message_Type);

  function Retrieve_Oldest_Message return Message_Type;
  function Total_Messages_Logged   return Natural;

end Message_Log_Package;
```

(Continued on next page)

```

package body Message_Log_Package is

    Total_Received : Natural := 0;

    procedure Log_Message (Incoming_Message : in Message_Type) is

        Interval : Duration;

    begin
        -- Fill log entry:
        Message_Log (Next_Entry_Location).Message_Part := Incoming_Message;
        Message_Log (Next_Entry_Location).Time_Received := Clock;
        Interval := Message_Log (Next_Entry_Location).Time_Received -
            Last_Time_Logged;
        Message_Log (Next_Entry_Location).Time_Since_Last := Interval;

        -- Update global variables:
        Last_Time_Logged :=
            Message_Log (Next_Entry_Location).Time_Received;
        Total_Received := Total_Received + 1;

        if Next_Entry_Location < Message_Log_Index_Subtype'Last then
            Next_Entry_Location := Next_Entry_Location + 1;
        else
            Next_Entry_Location := Message_Log_Index_Subtype'First;
        end if;
    end Log_Message;

    function Retrieve_Oldest_Message return Message_Type is
        Return_Location : Message_Log_Index_Subtype :=
            Next_Retrieval_Location;

    begin
        if Next_Retrieval_Location < Message_Log_Index_Subtype'Last then
            Next_Retrieval_Location := Next_Retrieval_Location + 1;
        else
            Next_Retrieval_Location := Message_Log_Index_Subtype'First;
        end if;
        return Message_Log (Return_Location).Message_Part;
    end Retrieve_Oldest_Message;

    function Total_Messages_Logged return Natural is
    begin
        return Total_Received;
    end Total_Messages_Logged;

end Message_Log_Package;

```

## DISCUSSION

In the original solution, the package interface was kept as simple as possible. The alternative solution illustrates the other extreme, with all declarations visible to the user (except for Total Received, since the Problem stated explicitly that this object should be inaccessible). This specification is much more cluttered; the user is given more information than necessary, inviting undesirable access to data. In addition, a change to the structure of the log would now involve recompilation of both body and specification, whereas the earlier version would require only recompilation of the body.

In order to move all declarations into the specification, the context clauses naming Calendar must be moved to beginning of the package. All associated entities are now available throughout Message\_Log\_Package, even though they need not be part of the package interface. In addition, a use clause has been added, eliminating the renaming declarations. The loss of clarity here is not substantial, since only one unit is being imported; however, it is conceivable that other units might be included later, at which point the origins of Time and Clock would become less obvious. Renaming declarations or expanded names provide documentation for all imported entities, requiring only a little extra writing.

The subprogram bodies are the same here as in the original solution, with the exception that Log\_Message now calls Clock rather than the renamed Current Time. The renaming declaration for Message\_Log (Next\_Entry\_Location) has also been removed, creating long variable references that decrease the readability of the statements.

The executable portion of the body has been removed, since the Clock function is visible when Last Time Logged is declared and can be used at that point to initialize the variable. In the original solution, the renaming declaration for Current Time could be moved up so that Last Time Logged could be initialized similarly; it was coded as shown to demonstrate one use of the executable part of a package.

This page intentionally left blank

## **Section 14**

### **EXCEPTIONS**

#### **14.1 Exceptions and Exception Handling**



This page intentionally left blank

## SECTION 14.1

### OBJECTIVE

To introduce exceptions and exception handling.

### TUTORIAL

In Ada, error situations that arise during program execution are called exceptions. For example, attempting to call a subprogram before the subprogram's body is known is an error situation which could occur at runtime. When these error situations occur, the implementation draws attention to it by raising an exception. When an exception is raised, normal program execution is abandoned. For example, consider the following procedure:

```
type Arr is array (Integer range <>) of Float;

with Text_IO; use Text_IO;
procedure Compute_And_Print_Average (X : in Arr) is

    Sum : Float := 0.0;
    Average : Float;
    package Fl_IO is new Float_IO (Float);
    use Fl_IO;

begin -- Compute_And_Print_Average

    for I in X'Range
    loop
        Sum := Sum + X(I);
    end loop;

    Average := Sum / X'Length;
    Put ("The average is ");
    Put (Average);
    New_Line;

end Compute_And_Print_Average;
```

If Compute And Print Average is called with a null array, X'Length is zero, causing computation of Average to raise an exception because division by zero is mathematically undefined. When this happens the last three lines (the IO statements) are not executed. Execution returns to the parent unit (i.e., the unit that it is contained in or that called it.)

In Ada there are several language-defined exceptions. They are: Data\_Error, Numeric\_Error, Program\_Error, Storage\_Error, and Tasking\_Error. These exceptions are raised automatically for language-defined reasons.

Data Error is raised when a range constraint, or index constraint is violated. For example:

```
X : Integer range 1 .. 10 := 20;
```

would raise Constraint\_Error.

Numeric\_Error is raised when a numeric operation is unable to provide an accurate numerical result. Specifically, Numeric\_Error is the exception raised by Compute\_And\_Print\_Average when X'Length is zero and division is attempted.

Program\_Error is raised when an attempt is made to call a program unit before the body is processed. For example, given

```
package Queue_Operations is

  type Queue is array (1 .. 10) of Natural;
  function Sort_Queue (Q : in Queue) return Queue;
  Standard_Queue : Queue := (5,6,3,2,9,8,7,1,10,4);
  Sorted_Queue   : Queue := Sort_Queue (Standard_Queue);

end Queue_Operations;
```

the call to Sort\_Queue in the initialization of Sorted\_Queue will raise Program\_Error because the body of Sort\_Queue has not been seen yet.

Program\_Error is also raised when a function is exited by some means other than a return statement. For example, in the following subprogram:

```
function Convert_True_To_Twenty (X : Boolean) return Integer is
begin -- Convert_True_To_Twenty

  if X then
    return 20;
  end if;

end Convert_True_To_Twenty;
```

Program\_Error is raised when the function is called with an actual value of False.

Program\_Error may also be raised when an expression uses a variable with an undefined value. For example, in Compute\_And\_Print\_Average, if not all the elements of the array have values assigned to them, Program\_Error could be raised. It should be noted that in this case an implementation is free not to raise an exception. Some implementations will and some will not.

Storage\_Error is raised when the available storage is insufficient.

Tasking\_Error is raised when an exception occurs during inter-task communication.

Text\_IO has predefined exceptions also. They are Status\_Error, User\_Error, Data\_Error, End\_Error, Layout\_Error, Mode\_Error, Name\_Error, and Device\_Error.

Ada allows programs to handle exceptions. To handle an exception is to perform some action in response to the raising of an exception. Package bodies, subprogram bodies, task bodies, and block statements have exception handlers as an optional part of their structure. The syntax for exception handlers is:

```
begin
:
:
exception
  when Exception Choice =>
    Sequence Of Statements;
  when Exception Choice =>
    Sequence Of Statements;
end;
```

The exception handlers appear just prior to the end of the program unit and after the keyword "exception". The "when Exception Choice => Sequence Of Statements" is called a handler. The Exception Choice names the exception to which the handler applies. The Sequence Of Statements are the actions to be performed when the exception is handled. There can be many handlers between the word exception and the end of the program unit.

In the following, the exception Data\_Error, raised by incorrect data being entered, is handled.

```
with Text_IO; use Text_IO;
procedure Compute_Average is
  X : array (Integer range 1 .. 10) of Float;
  Sum : Float := 0.0;
  Average : Float;
  package Fl_IO is new Float_IO (Float);
  use Fl_IO;

  Begin -- Compute_Average
    for I in X'Range
    loop
      Get (X(I));
      Sum := Sum + X(I);
    end loop;

    Average := Sum / X'Length;
    Put ("The average is ");
    Put (Average);
    New_Line;

  exception
    when Data_Error =>
      Put_Line ("Incorrect Data Entered.");
    when others =>
      Put_Line ("Some error - Program incomplete");
  end Compute_Average;
```

Note the use of "others" in the handler. This handler applies to all other possible exceptions. Like "others" in case statements, this choice must be in the last handler and must appear by itself.

If several exceptions are handled in the same manner, the handlers can be written:

```
when Exception_1 | Exception_2 =>  
    Initiate_Recovery_Strategy;
```

Ada allows user-defined exceptions. The format for defining an exception is:

```
Exception Name: exception;
```

Below are examples of exception declarations.

```
Divide_By_Zero : exception;  
Sensor_Malfunction : exception;  
Plane_Too_Low : exception;  
Queue_Full : exception;
```

Ada also provides a mechanism to explicitly raise exceptions. This mechanism is the raise statement. The syntax for the raise statement is:

```
raise Exception Name;
```

The following statements are examples of raise statements.

```
raise Division_By_Zero;  
raise Sensor_Malfunction;  
raise Plane_Too_Low;  
raise Queue_Full;
```

In the package below, the user-defined exceptions are used to notify the user of the package when an attempt is made to Pop an item off an empty stack or to Push an item on a full stack.

```

package Stack_Manager is

    procedure Push (X: Integer);
    procedure Pop returns Integer;
    Stack_Full, Stack_Empty : exception;

end Stack_Manager;

package body Stack_Manager is

    type Stack is array (1 .. 100) of Integer;
    Top : Integer := 1;
    Int_Stack : Stack;

    procedure Push (X: Integer) is
    begin -- Push

        if Top > 100 then
            raise Stack_Full;
        else
            Int_Stack (Top) := X;
            Top := Top + 1;
        end if;

    end Push;

    function Pop return Integer is
    begin -- Pop

        if Top = 1 then
            raise Stack_Empty;
        else
            Top := Top - 1;
            return Int_Stack (Top);
        end if;

    end Pop;

end Stack_Manager;

```

Note that the user of Stack\_Manager is aware that these exceptions can be raised because they appear in the visible part of the package specification. The user can write exception handlers for these exceptions just as if they were language-defined exceptions.

One other point of interest - the language-defined exceptions may also be explicitly raised by the raise statement. This practice, although legal, should be avoided. Raising a predefined exception in a place where it will be raised anyway is a duplication of effort, yours and the compiler's. Raising a predefined exception where it does not apply is sure to give the maintainer headaches. In general, it is a good idea to name the exceptions you specifically wish to guard against.

Recall that when an exception is raised, normal execution is abandoned and if a handler for that exception is present, it is executed before control is returned to the parent unit. In some cases, this is unacceptable (especially in real-time applications). Here you want to handle the exception and continue processing as if nothing has happened.

The following simple example illustrates the disadvantage of completing execution when an exception is raised.

```
with Text_IO; use Text_IO;
procedure Sum_Digits is

    subtype Input_type is Integer range 100 .. 999;
    Sum : Integer := 0;
    Input : Integer;
    package Input_IO is new Integer_IO (Input_type);
    package Int_IO is new Integer_IO (Integer);

begin -- Sum_Digits

    Put_Line ("Enter your three digit number");
    Input_IO.Get (Input);
    for I in 1 .. 3
    loop
        Sum := Sum + (Input mod 10);
        Input := Input / 10;
    end loop;
    Put ("The sum is ");
    Int_IO.Put (Sum);

exception
    when Data_Error =>
        Put_Line ("Must enter a three digit number.");
    when others =>
        Put_Line ("Some error - Program incomplete.");
end Sum_Digits;
```

The Get statement will raise Data\_Error when anything but a three digit number is entered, which is good because the program will not work correctly for anything but a three digit number. However, in this case, when the exception is raised the program's execution completes. It is undesirable for a poor typist to have to re-execute the program every time he or she makes a mistake.

To prevent this situation, the statement which is likely to raise an exception can be isolated so the exception can be raised and handled locally. Block statements are used to isolate statements which are known to raise exceptions.

The syntax for the block statement is:

```
[declare
    Sequence Of Declarations;
begin
    Sequence Of Statements;
[exception
    Handlers;]
end;
```

Note that the declarative part and the exception part are optional.

Using a block statement, the above program can be written:

```
with Text_IO; use Text_IO;
procedure Sum_Digits is

    subtype Input_type is Integer range 100 .. 999;
    Sum : Integer := 0;
    Input : Integer;
    package Input_IO is new Integer_IO (Input_type);
    package Int_IO is new Integer_IO (Integer);

begin -- Sum_Digits

    Put_Line ("Enter your three digit number");

    loop
        begin
            Input_IO.Get (Input);
            exit;
        exception
            when Data_Error =>
                Put_Line ("Entry Error");
                Put_Line ("Re-enter your three digit number");
        end;
    end loop;

    for I in 1 .. 3
    loop
        Sum := Sum + (Input mod 10);
        Input := Input / 10;
    end loop;
    Put ("The sum is ");
    Int_IO.Put (Sum);

    exception
        when others =>
            Put_Line ("Some error - Program incomplete");

end Sum_Digits;
```



Now the Get statement is contained inside a block statement which handles the exception locally. Note the addition of the loop statement which allows the poor typist to try as many times as necessary to enter his or her three digit number. In this version, Data\_Error can be raised and program execution will continue after the situation is corrected.

### PROBLEM

Consider a system where messages are sent over a complex network of message nodes. In order to avoid multiple accessing of the same message node, the node is seized and locked during message transmission.

All the message nodes have approximately the same form, shown below.

```
begin
  loop
    if Receiving_A_Message then
      Process_Message;
    elsif Sending_A_Message then
      Seize (Message_Node);
      Send_Message (Message);
      Free (Message_Node);
    end if;
  end loop;
end;
```

This solution, while prohibiting dual accessing, has created another problem: whenever any kind of error situation occurs during message transmission, the receiving node remains forever locked, requiring operator intervention to free the node.

Modify this algorithm so that when anything goes wrong during message transmission, the message node is freed and the operator is notified that the message was not delivered.

This page intentionally left blank

## SOLUTION TO EXERCISE 14.1

### SOLUTION

```
begin
  loop
    if Receiving_A_Message then
      Process_Message;
    elsif Sending_A_Message then
      Seize (Message_Node);
      begin
        Send_Message (Message);
      exception
        when others =>
          Notify_Operator;
      end;
      Free (Message_Node);
    end if;
  end loop;
end;
```

### RATIONALE FOR SOLUTION

The addition of the block statement with the exception handler isolates or traps any exception that is raised during message transmission, and notifies the operator that the message was not sent.

Now, when exceptions are raised, they are handled, and the message node is freed as if the message had been successfully sent.

Note the use of "others". This guarantees that any exception raised is handled. In this case, "others" is useful because the full range of possible exceptions is not known (Send Message may have declared its own) and the same handling is done regardless of the exception.

### ALTERNATIVE SOLUTION

```
begin
  loop
    if Receiving_A_Message then
      Process_Message;
    elsif Sending_A_Message then
      Seize (Message_Node);
      Send_Message (Message);
      Free (Message_Node);
    end if;
  end loop;
exception
  when others =>
    Free (Message_Node);
    Notify_Operator;
end;
```

### DISCUSSION

Both solutions free the message node and notify the operator upon any exception being raised. However, the first solution is superior for two reasons: First, unlike the preferred solution which handles the exception inside the loop, the alternate solution handles the exception outside the loop. This causes the execution of the message node to terminate after the exception is handled. This is clearly undesirable in a realtime system that depends on the message nodes being active.

Second, the alternate solution assumes that only Send\_Message can raise exceptions and that the same handling is done no matter what raised the exception. This is hardly an accurate assumption. Suppose Process\_Message, Seize, or Free raised the exception. Freeing the message node is hardly an appropriate handler for an exception that is raised in any of these subprograms. Process\_Message has nothing to do with freeing a message node, and it is reasonable to assume that if Seize raised an exception nothing needs to be freed, and if Free raised an exception the last thing you would want to do is call it again.

The first solution assures the exception was raised in Send\_Message and allows other handlers to be written for any other problem areas that may be identified.

## **Section 15**

### **INPUT/OUTPUT**

#### **15.1 Package Text\_IO**

This page intentionally left blank

## EXERCISE 15.1

### OBJECTIVE

To introduce the package Text\_IO.

### TUTORIAL

Ada does not have built-in features for handling input and output. The capability is, instead, contained in a package and is subsequently "withed" and "used". This format allows greater flexibility in redefining special input/output subprograms.

The package Text\_IO provides procedures and functions for input and output of text. In order to operate on external files, the file must be created by using the procedure Create, or opened using the procedure Open. An external file is anything outside the program that can produce a character for input or accept a character for output. On the other hand, an internal file is an object within the program that is able to be associated with an external file.

There are five basic file commands in Text\_IO: Create, Open, Close, Delete, and Reset. Briefly, Create creates and opens a new external file; Open establishes the link between an internal and an existing external file; Close severs that link; Delete deletes the external file; Reset resets parameters so subsequent I/O starts at the beginning of the file.

The procedure declaration for Create is as follows:

```
procedure Create (File : in out File_Type;  
                 Mode : in File_Mode := Default_Mode;  
                 Name : in String := "";  
                 Form : in String := "");
```

where File is the internal file to be associated with the newly created external file, Mode defines the file as an input file (In File) or an output file (Out File), Name identifies the external file, and Form specifies the file form (used for permissions, size limits, etc.).

For instance, in the following section of code,

```
R_File : File_Type;  
Create (R_File, Out_File, "Data");
```

the Create statement creates a file called Data which is initially an Out File, meaning information can only be written to it. The internal file, in this example, is named R\_File, and the external file is named "Data". The Form has the default value specified by the implementation. Note that Create both creates and opens a file.



The specification for Open is as follows:

```
procedure Open (File : in out File_Type;  
               Mode : in File_Mode;  
               Name : in String;  
               Form : in String := "");
```

Open associates an internal file with an existing external file. The formal parameters are the same as in the procedure Create. Notice, however, that there is no default for name, and a null string as an actual parameter for the name would raise the exception Name\_Error. Also, it is illegal to open an internal file that has already been opened (or created).

The Close command,

```
procedure Close (File : in out File_Type);
```

severs the association between the internal and external files. The formal parameter File is the internal file to be closed.

Delete,

```
procedure Delete (File : in out File_Type);
```

deletes the external file associated with File. The internal file must be open when the procedure is called.

And finally Reset,

```
procedure Reset (File : in out File_Type;  
                Mode : in out File_Mode);
```

commences reading from or writing to a file at the beginning of the file. The parameter Mode allows the mode of the file to be changed when it is reset. The internal file must be open when Reset is called.

There are four functions, corresponding loosely to the parameters of Create and Open, that are used to obtain information about a file. They are:

```
function Mode (File : in File_Type) return File_Mode;  
function Name (File : in File_Type) return String;  
function Form (File : in File_Type) return String;  
function Is_Open (File : in File_Type) return Boolean;
```

Their actions are as their names suggest. Mode returns the mode of File. Name returns the name of File. Form returns the form of File. And Is\_Open returns a Boolean value indicating whether or not File is open.

Default files are files that are used when no file is specified. Standard files are open at the beginning of program execution. The current input file and the current output file refer to the current default files. They are initially the standard input and standard output files but may be changed. The commands dealing with default files will not be discussed here. Briefly, they are:

```
procedure Set_Input (File : in File_Type);
procedure Set_Output (File : in File_Type);
function Standard_Input return File_Type;
function Standard_Output return File_Type;
function Current_Input return File_Type;
function Current_Output return File_Type;
```

These functions are intended to be used in conjunction with other I/O routines; they may be used as parameters to other subprograms.

Information pertaining to the lines and pages of a file may be obtained using the following functions:

```
function Line_Length (File : File_Type) return Count;
function Page_Length (File : File_Type) return Count;
function End_Of_Line (File : File_Type) return Boolean;
function End_Of_Page (File : File_Type) return Boolean;
function End_Of_File (File : File_Type) return Boolean;
function Col (File : in File_Type) return Positive_Count;
function Line (File : in File_Type) return Positive_Count;
function Page (File : in File_Type) return Positive_Count;
```

where type Count is an integer with a lower bound of zero (upper bound implementation defined).

Line\_Length and Page\_Length return the current line length and page lengths, respectively, of File. These two functions must be applied to an output file (meaning a file that is written to). End\_of\_Line, End\_of\_Page, and End\_of\_File, on the other hand, must be applied to an Input file (a file that is read from). These functions return a Boolean value indicating whether the end of the current line, page, or file has been encountered. These five functions may be called without a parameter, in which case they operate on the current default input or output file.

Col, Line, and Page return the column, line, and page, of File. They may be applied to an input file or an output file. These three functions may also be called without a parameter, in which case they are applied to the current output file. For all of the functions above, the specified file must be open.

Operations to set the line or page length of a specific file are:

```
procedure Set_Line_Length (File : in File_Type;  
                           To   : in Count);  
procedure Set_Page_Length (File : in File_Type;  
                           To   : in Count);
```

The formal parameter To represents the intended length. Both of these procedures must be applied to a file of mode Out\_File.

Procedures to increment the line or page of an output file are:

```
procedure New_Line (File      : in File_Type;  
                   Spacing   : in Positive_Count := 1);  
procedure New_Page (File      : in File_Type);
```

Similar procedures for skipping lines and pages of input files:

```
procedure Skip_Line (File      : in File_Type;  
                   Spacing   : in Positive_Count := 1);  
procedure Skip_Page (File      : in File_Type);
```

Skip\_Line and Skip\_Page, for an output file, recommence writing at the next line or page. For an input file, they read and discard all characters until the beginning of the next line or page.

Finally, there are Set\_Col and Set\_Line.

```
procedure Set_Col (File      : in File_Type;  
                  To        : in Positive_Count);  
procedure Set_Line (File      : in File_Type;  
                  To        : in Positive_Count);
```

For an input file, Set\_Col and Set\_Line read and discard characters, line terminators, and page terminators, until the column number or line number of the next character to be read corresponds to the specified parameter To. For an output file, they write spaces or line terminators until the column number or line number equals the specified value.

Set\_Line\_Length, Set\_Page\_Length, New\_Line, New\_Page, Set\_Col, and Set\_Line may all be called with the parameter File omitted, in which case they are applied to the current default output file. Skip\_Line and Skip\_Page, apply to the current default input file when the file parameter is omitted.

Get and Put procedures exist for many types: character, string, integer, fixed point, floating point, predefined enumeration, and user-defined enumeration. To access the packages containing Get and Put procedures for these different types the programmer must first import Text\_IO, and then instantiate the appropriate package (for example Float\_IO). But we are only concerned here with Text\_IO, which supports Get and Put for characters and strings.

Like most of the subprograms mentioned above, Get and Put need not be called with a specified file. If the file is omitted, the appropriate current default file is used. The procedure specifications for character and string Get are:

```
procedure Get (File : in File_Type;  
              Item : out Character);  
  
procedure Get (File : in File_Type;  
              Item : out String);  
  
procedure Get_Line (File : in File_Type;  
                  Item : out String;  
                  Last : out Natural);
```

Last in Get\_Line is the index value of the last character in the string that is read. When Get is called with a String parameter characters are read, ignoring line and page terminators, until the string is full. Get\_Line reads until a line terminator is the next character to be read, or until the end of the string, whichever is sooner.

The specifications for character and string Put procedures are:

```
procedure Put (File : in File_Type;  
              Item : in Character);  
  
procedure Put (File : in File_Type;  
              Item : in String);  
  
procedure Put_Line (File : in File_Type;  
                  Item : in String);
```

When the parameter Item in a Put command is a string the procedure outputs each character in the string, even if it spans over several lines. Put\_Line, however, outputs the string on a single line, followed by a line terminator.

This page intentionally left blank

### PROBLEM

In a file named Angola\_Statistics there is the following data about that country: official name, capital, official language, currency, head of state, predominant ethnic group, and major export. This information is stored as strings, one category per line. Write a program that will read the contents of Angola\_Statistics and print it so that it appears in the following format:

```
Nation's Name:  People's Republic of Angola
Capital:  Luanda
Official Language:  Portuguese
Currency:  Kwanza
Head of State:  President Jose Eduardo dos Santos
Predominant Ethnic Group:  Bantu
Major Export:  Iron ore
```

This page intentionally left blank

## SOLUTION TO EXERCISE 15.1

### SOLUTION

```
with Text_IO; use Text_IO;
procedure Print_National_Statistics is
    Nations_Name,
    Capital,
    Language,
    Currency,
    Head_Of_State,
    Predominant_Ethnic_Group,
    Major_Export      : String (1 .. 30);
    National_Statistics : File_Type;
    Nations_Name_Last,
    Capital_Last,
    Language_Last,
    Currency_Last,
    Head_Of_State_Last,
    Ethnic_Group_Last,
    Major_Export_Last      : Natural;
begin
    Open (National_Statistics, In_File, "Angola_Statistics");
    Set_Input (National_Statistics);

    Get_Line (Nations_Name, Nations_Name_Last);
    Get_Line (Capital, Capital_Last);
    Get_Line (Language, Language_Last);
    Get_Line (Currency, Currency_Last);
    Get_Line (Head_Of_State, Head_Of_State_Last);
    Get_Line (Predominant_Ethnic_Group, Ethnic_Group_Last);
    Get_Line (Major_Export, Major_Export_Last);

    Set_Input (Standard_Input);
    Close (National_Statistics);

    Put      ("Nation's Name: ");
    Put_Line (Nations_Name (1 .. Nations_Name_Last));
    Put      ("Capital: ");
    Put_Line (Capital (1.. Capital_Last));
    Put      ("Official Language: ");
    Put_Line (Language (1 .. Language_Last));
    Put      ("Currency: ");
    Put_Line (Currency (1.. Currency_Last));
    Put      ("Head of State: ");
    Put_Line (Head_Of_State (1 .. Head_Of_State_Last));
    Put      ("Predominant Ethnic Group: ");
    Put_Line (Predominant_Ethnic_Group (1 .. Ethnic_Group_Last));
    Put      ("Major Export: ");
    Put_Line (Major_Export (1 .. Major_Export_Last));

end Print_National_Statistics;
```



## RATIONALE FOR SOLUTION

The solution declares an object of type string for each of the seven categories of information in the problem. The appropriate strings are then read into these objects, and subsequently printed after their corresponding label.

Before the first line of the procedure the package Text\_IO is "withed" and "used", making the text input and output routines available. It is necessary to "with" Text\_IO in order to read from and write to files. "Use", on the other hand, is not necessary, but as explained in Exercise 12.1, functionally places the I/O routines in Text\_IO within the immediate scope of the program, and allows us to write, for instance,

```
Get (Something);
```

instead of

```
Text_IO.Get (Something);
```

In the declarative part of the procedure, note National\_Statistics which is declared an object of type File\_Type. It is the internal file associated with Angola\_Statistics.

The external file Angola\_Statistics is then opened, using the file command "Open". The three parameters given with Open are National\_Statistics, the name of the internal file, the mode In file, specifying that the file is for reading rather than writing, and Angola\_Statistics, the name of the existing external file.

National\_Statistics is then made the default input file by the Set\_Input command. This allows the subsequent Gets not to specify the file explicitly.

The information is successively read from Angola\_Statistics into each of the seven declared strings. Notice that Get\_Line is used. It takes three parameters: the name of the internal file, the object into which the information goes, and an index for the last character put into the string.

After the reading, the default input file is set back to Standard\_Input and the file is closed.

The statistics are printed using Put and Put\_line. The latter is the same as a regular Put with an additional new-line printed. First, the string label and colon is printed using Put, so that the succeeding string is printed on the same line. Put\_Line prints the information read from Angola\_Statistics followed by a new-line. There is no file parameter given here for either of these functions because they refer to the standard default output file.

Notice that the value actually Put is the slice actually read in. This was the reason for keeping track of the index of each of the seven values read. If these values had not been saved, the part of the array that wasn't assigned to would be undefined (ie. possible full on junk).

### ALTERNATIVE SOLUTION

```
with Text_IO; use Text_IO;
procedure Print_National_Statistics is

    Nations_Name,
    Capital,
    Language,
    Currency,
    Head_Of_State,
    Predominant_Ethnic_Group,
    Major_Export : String (1 .. 30);
    National_Statistics : File_Type;

begin

    Open (National_Statistics, In_File, "Angola_Statistics");

    Get (National_Statistics, Nations_Name);
    Get (National_Statistics, Capital);
    Get (National_Statistics, Language);
    Get (National_Statistics, Currency);
    Get (National_Statistics, Head_Of_State);
    Get (National_Statistics, Predominant_Ethnic_Group);
    Get (National_Statistics, Major_Export);

    Close (National_Statistics);

    Put ("Nation's Name: ");
    Put (Nations_Name);
    New_Line;
    Put ("Capital: ");
    Put (Capital);
    New_Line;
    Put ("Official Language: ");
    Put (Language);
    New_Line;
    Put ("Currency: ");
    Put (Currency);
    New_Line;
    Put ("Head of State: ");
    Put (Head_Of_State);
    New_Line;
    Put ("Predominant Ethnic Group: ");
    Put (Predominant_Ethnic_Group);
    New_Line;
    Put ("Major Export: ");
    Put (Major_Export);
    New_Line;

end Print_National_Statistics;
```

## DISCUSSION

In the alternate solution, Get is used instead of Get\_Line to read information from Angola Statistics into the seven declared strings. This requires that these strings be exactly 30 characters long. Get, when given a string of 30 characters, will read exactly thirty characters. While this alleviates the need for last, it requires that the string be padded with blanks. In a system where storage is tight, this solution is clearly undesirable.

Note also that this solution does not change the default input file. This is a matter of preference, either is acceptable. However, when only one input file is involved, it can cause no confusion to change the default input files.

The alternate solution also uses Puts and New\_Lines instead of Put\_Lines. Both methods work, but the former solution is more succinct.

## GLOSSARY

access type	A type which defines a set of values used to designate objects created by allocators.
actual parameters	The value or variable associated with a formal parameter in a subprogram call. See parameter.
aggregate	A grouping of values for a composite type.
allocator	An expression which, when evaluated, creates an object and returns a new access value designating the object.
anonymous type	An unnamed type.
array type	A type containing components of the same subtype.
attribute	A property of a type, object, or subprogram.
block statement	A compound statement with optional declarative part and exception handler. See statement.
compilation	A succession of compilation units submitted to the compiler.
compilation unit	A library unit or a secondary unit.
component	An object that is part of a larger object.
composite type	A type which contains components. A record or array type.
conditional exit	An exit statement with a when clause.
constant	An object whose value cannot change during program execution.
context clause	A sequence of with clauses and/or use clauses.
declarative region	The region of text that contains a declaration. The region following "is" and prior to "begin" in subprograms and packages, and the region following "declare" and prior to "begin" in blocks.
dereferencing	Process of referring to an allocated variable.

discrete type	A type which has an ordered set of discrete values. Can be an integer type or an enumeration type.
discriminant	A distinguished component of an object of a record type, used in the definition of other components.
enumeration type	A discrete type whose values are represented by identifiers or character literals.
exception	A runtime error.
formal parameters	Parameters defined in subprogram definition. See parameters.
generic	A template.
iteration scheme	The for clause or the while clause of loop statements.
library unit	A separate compilation unit: a subprogram specification or body, or a package specification.
loop statement	A compound statement whose execution results in the repetition of a sequence of statements.
mode	Manner of use of formal parameters, i.e., "in," "out," or "in out."
notation	Named, where the aggregate or parameter components are listed by name.
	Positional, where the aggregate or parameter components are listed in order of occurrence.
object	Variable or constant.
overload	Allow identifiers to have several alternative meanings.
package	Group of logically related entities.
parameters	Named entities defined in a subprogram declaration. Used as an interface to pass values between the caller and the subprogram.

private type	A type whose structure and set of values are defined but not available to the user of the type.
program library	A file containing information about the compilation units.
program unit	A subprogram, task, package, or generic unit. The unit consists of a specification and a body.
record type	A composite type of unlike components.
scope	Region of program text in which an entity is potentially visible.
secondary unit	The body of a library unit or a subunit.
slice	Part of an array object.
specification	Definition of the interface of an entity.
statement	Actions performed during program execution.
static	Value known at compile time.
string type	An array type of characters.
stub	See subunit.
subprogram	A procedure or function.
subtype	A subset of a type's set of values.
subunit	Separately compiled unit: a package or subprogram body
type	A set of values and operations on those values.
variable	An object whose value can change during program execution.

This page intentionally left blank

CROSS REFERENCE TO  
THE ADA LANGUAGE REFERENCE MANUAL

Unit 1 -- Introduction and Overview of the Ada Language

context clause	-- LRM Chapter 10.1.1
file	-- LRM Chapter 14.1
I/O	-- LRM Chapter 14.3
loop statement	-- LRM Chapter 5.5
procedure	-- LRM Chapter 6.1
package	-- LRM Chapter 7.1

Unit 2 -- Introduction to Program Units

actual parameter	-- LRM Chapter 6.4
array type	-- LRM Chapter 3.6
compilation unit	-- LRM Chapter 10.1
formal parameter	-- LRM Chapter 6.2
function	-- LRM Chapter 6.5
function body	-- LRM Chapter 6.3
function specification	-- LRM Chapter 6.1
generics	-- LRM Chapter 12.1
library unit	-- LRM Chapter 10.1
package	-- LRM Chapter 7.1
package body	-- LRM Chapter 7.3
package specification	-- LRM Chapter 7.2
parameters	-- LRM Chapter 6.1



## Unit 2 -- (Continued)

parameter modes	-- LRM Chapter 6.2
procedure	-- LRM Chapter 6.1
procedure body	-- LRM Chapter 6.3
procedure specification	-- LRM Chapter 6.1
program library	-- LRM Chapter 10.1, 10.4
return statement	-- LRM Chapter 5.8
separate	-- LRM Chapter 10.2
stub	-- LRM Chapter 10.2
subprograms	-- LRM Chapter 6.1
subunit	-- LRM Chapter 10.2
tasks	-- LRM Chapter 9.1
type	-- LRM Chapter 3.3

## Unit 3 -- Lexical Elements

character set	-- LRM Chapter 2.1
comment	-- LRM Chapter 2.7
constant	-- LRM Chapter 3.2.1
delimiters	-- LRM Chapter 2.2
identifiers	-- LRM Chapter 2.3
lexical elements	-- LRM Chapter 2.2
numeric literals	-- LRM Chapter 2.4
reserve words	-- LRM Chapter 2.9
type	-- LRM Chapter 3.3
variable	-- LRM Chapter 3.2.1

#### Unit 4 -- Introduction to Data

assignment statement	-- LRM Chapter 5.2
expressions	-- LRM Chapter 4.4
floating point type	-- LRM Chapter 3.5.7
initialization	-- LRM Chapter 3.2.1
integer literals	-- LRM Chapter 2.4
membership operations	-- LRM Chapter 4.5.2
numeric operations	-- LRM Chapter 4.5.3
objects	-- LRM Chapter 3.2
operator precedence	-- LRM Chapter 4.5
relational operations	-- LRM Chapter 4.5.2
range constraint	-- LRM Chapter 3.3
real literals	-- LRM Chapter 2.4
type conversion	-- LRM Chapter 4.6
type Integer	-- LRM Chapter 3.5.4
type	-- LRM Chapter 3.3
unary operations	-- LRM Chapter 4.5.4
variable	-- LRM Chapter 3.2.1

## Unit 5 -- Enumeration : Types and Control Structures

boolean type	-- LRM Chapter 3.5.3
case statement	-- LRM Chapter 5.4
choice	-- LRM Chapter 3.7.3
discrete range	-- LRM Chapter 3.6.1
discrete type	-- LRM Chapter 3.5
enumeration literals	-- LRM Chapter 3.5.1
enumeration types	-- LRM Chapter 3.5.1
floating point types	-- LRM Chapter 3.5.7
if statement	-- LRM Chapter 5.3
integer types	-- LRM Chapter 3.5.4
variables	-- LRM Chapter 3.2.1

## Unit 6 -- Numerics

constant	-- LRM Chapter 3.2.1
fixed point types	-- LRM Chapter 3.5.9
floating point types	-- LRM Chapter 3.5.7
integer types	-- LRM Chapter 3.5.4
numeric types	-- LRM Chapter 3.5.6
objects	-- LRM Chapter 3.2.1
range constraint	-- LRM Chapter 3.3.2
subtype	-- LRM Chapter 3.3, 3.3.2

## Unit 7 -- Advanced Features of Scalar Types

alternatives	-- LRM Chapter 5.4
attributes	-- LRM Chapter 3.5.5
base type	-- LRM Chapter 3.3
boolean expressions	-- LRM Chapter 3.5.3
case statement	-- LRM Chapter 5.4
fixed point types	-- LRM Chapter 3.5.9
floating point types	-- LRM Chapter 3.5.7
integer literal	-- LRM Chapter 2.4
logical and	-- LRM Chapter 4.5.1
logical operators	-- LRM Chapter 4.5.1
logical or	-- LRM Chapter 4.5.1
membership operators	-- LRM Chapter 4.5.2
object	-- LRM Chapter 3.2
scalar types	-- LRM Chapter 3.5
short circuit control form	-- LRM Chapter 4.5.1
type	-- LRM Chapter 3.3

## Unit 8 -- Array Types and Iterative Control Structures

anonymous array type	-- LRM Chapter 3.6
array aggregate	-- LRM Chapter 4.3.2
array object	-- LRM Chapter 3.6
array type	-- LRM Chapter 3.6
attributes	-- LRM Chapter 3.6.2
boolean value	-- LRM Chapter 3.5.3
catenation operator	-- LRM Chapter 4.5.3
character type	-- LRM Chapter 3.5.2
component subtype	-- LRM Chapter 3.6
composite types	-- LRM Chapter 3.3
condition	-- LRM Chapter 5.3
conditional exit	-- LRM Chapter 5.7
constant	-- LRM Chapter 3.2.1
discrete range	-- LRM Chapter 3.6.1
discrete type	-- LRM Chapter 3.3
enumeration type	-- LRM Chapter 3.5.1
exit statement	-- LRM Chapter 5.7
floating point type	-- LRM Chapter 3.5.7
for loop	-- LRM Chapter 5.5
formal parameter	-- LRM Chapter 6.2
function body	-- LRM Chapter 6.3
goto statement	-- LRM Chapter 5.9
index constraint	-- LRM Chapter 3.6.1

## Unit 8 -- (Continued)

index subtype	-- LRM Chapter 3.6
integer type	-- LRM Chapter 3.5.4
iteration scheme	-- LRM Chapter 5.5
label	-- LRM Chapter 5.9
loop parameter	-- LRM Chapter 5.5
loop statement	-- LRM Chapter 5.5
membership operator	-- LRM Chapter 4.5.2
named notation	-- LRM Chapter 4.3
null range	-- LRM Chapter 3.6.1
numeric literal	-- LRM Chapter 2.4
others	-- LRM Chapter 3.7.3
positional notation	-- LRM Chapter 4.3
range constraint	-- LRM Chapter 3.5
reverse	-- LRM Chapter 5.5
slice	-- LRM Chapter 3.6.2
static	-- LRM Chapter 4.9
string literal	-- LRM Chapter 2.6
string I/O	-- LRM Chapter 14.3
subaggregates	-- LRM Chapter 4.3
subtype	-- LRM Chapter 3.3, 3.3.2
unconstrained array	-- LRM Chapter 3.6
while loop	-- LRM Chapter 5.5

## Unit 9 -- Record types

array type	-- LRM Chapter 3.6
components	-- LRM Chapter 3.6
composite type	-- LRM Chapter 3.3
constant	-- LRM Chapter 3.2.1
discriminant	-- LRM Chapter 3.7.1
discriminant constraint	-- LRM Chapter 3.7.2
enumeration type	-- LRM Chapter 3.5.1
named notation	-- LRM Chapter 4.3
positional notation	-- LRM Chapter 4.3
others	-- LRM Chapter 3.7.3
range constraint	-- LRM Chapter 3.5
record aggregate	-- LRM Chapter 4.3.1
record objects	-- LRM Chapter 3.7
record types	-- LRM Chapter 3.7
unconstrained array type	-- LRM Chapter 3.6
variant	-- LRM Chapter 3.7.3

## Unit 10 -- Access Types

access types	-- LRM Chapter 3.8
allocator	-- LRM Chapter 4.8
array type	-- LRM Chapter 3.6
null value	-- LRM Chapter 3.8
record type	-- LRM Chapter 3.7
slice	-- LRM Chapter 3.6.2

## Unit 11 -- Program Structure and Separate Compilation

compilation	-- LRM Chapter 10.1
function	-- LRM Chapter 6.5
library unit	-- LRM Chapter 10.1
package body	-- LRM Chapter 7.3
package declaration	-- LRM Chapter 7.2
package specification	-- LRM Chapter 7.2
procedure	-- LRM Chapter 6.1
program library	-- LRM Chapter 10.1, 10.4
secondary unit	-- LRM Chapter 10.1
separate	-- LRM Chapter 10.2
stub	-- LRM Chapter 10.2
subprogram body	-- LRM Chapter 6.3
subprogram declaration	-- LRM Chapter 6.1
subunit	-- LRM Chapter 10.2
use clause	-- LRM Chapter 8.4
with clause	-- LRM Chapter 10.1.1

## Unit 12 -- Using Library Units

array type	-- LRM Chapter 3.6
assignment statement	-- LRM Chapter 5.2
block name	-- LRM Chapter 5.6
block statement	-- LRM Chapter 5.6
declarative region	-- LRM Chapter 8.1
direct visibility	-- LRM Chapter 8.3



## Unit 12 -- (Continued)

discrete type	-- LRM Chapter 3.3
entry families	-- LRM Chapter 9.5
enumeration literals	-- LRM Chapter 3.5.1
enumeration type	-- LRM Chapter 3.5.1
equality operators	-- LRM Chapter 4.5.2
exceptions	-- LRM Chapter 11.1
extended scope	-- LRM Chapter 8.2
formal parameters	-- LRM Chapter 6.2
full scope	-- LRM Chapter 8.2
generic formal type	-- LRM Chapter 12.1.2
generic unit	-- LRM Chapter 12.1
hiding	-- LRM Chapter 8.3
immediate scope	-- LRM Chapter 8.2
instantiation	-- LRM Chapter 12.3
iterative scheme	-- LRM Chapter 5.5
label	-- LRM Chapter 5.9
limited private types	-- LRM Chapter 7.4.4
loop name	-- LRM Chapter 5.5
loop parameter	-- LRM Chapter 5.5
loop statement	-- LRM Chapter 5.5
membership operators	-- LRM Chapter 4.5.2
package body	-- LRM Chapter 7.3
package declaration	-- LRM Chapter 7.2

## Unit 12 -- (Continued)

private types	-- LRM Chapter 7.4.1
object	-- LRM Chapter 3.2
operator overloading	-- LRM Chapter 6.7
overloading	-- LRM Chapter 6.6
record components	-- LRM Chapter 3.7
record type	-- LRM Chapter 3.7
scope	-- LRM Chapter 8.2
subprogram body	-- LRM Chapter 6.3
subprogram declaration	-- LRM Chapter 6.1
subtype	-- LRM Chapter 3.3, 3.3.2
tasks	-- LRM Chapter 9.1
type	-- LRM Chapter 3.3
use clause	-- LRM Chapter 8.4
visibility	-- LRM Chapter 8.3
visibility by selection	-- LRM Chapter 8.3

## Unit 13 -- Packages

body stub	-- LRM Chapter 10.2
context clause	-- LRM Chapter 10.1.1
discriminant	-- LRM Chapter 3.7.1
expanded name	-- LRM Chapter 4.1.3
package	-- LRM Chapter 7.1
package body	-- LRM Chapter 7.3

### Unit 13 -- (Continued)

package specification	-- LRM Chapter 7.2
private part	-- LRM Chapter 7.2
private type	-- LRM Chapter 3.3, 7.4
renaming declaration	-- LRM Chapter 8.5
subprogram body	-- LRM Chapter 6.3
visible part	-- LRM Chapter 7.2

### Unit 14 -- Exceptions

block statement	-- LRM Chapter 5.6
case statement	-- LRM Chapter 5.4
exception	-- LRM Chapter 11.1
exception handler	-- LRM Chapter 11.2
function	-- LRM Chapter 6.5
I/O exceptions	-- LRM Chapter 14.4
language defined exceptions	-- LRM Chapter 11.1
others	-- LRM Chapter 3.7.3
package body	-- LRM Chapter 7.3
package specification	-- LRM Chapter 7.2
raise statement	-- LRM Chapter 11.3
return statement	-- LRM Chapter 5.8
subprogram	-- LRM Chapter 6.1
subprogram body	-- LRM Chapter 6.3
task body	-- LRM Chapter 9.1

## Unit 15 -- Input/Output

default files	-- LRM Chapter 14.3.2
external file	-- LRM Chapter 14.1
file	-- LRM Chapter 14.1
file commands	-- LRM Chapter 14.2.1
file mode	-- LRM Chapter 14.1
get and put commands	
for character and string	-- LRM Chapter 14.3.6
for integer	-- LRM Chapter 14.3.7
for fixed and float	-- LRM Chapter 14.3.8
for enumeration	-- LRM Chapter 14.3.9
internal file	-- LRM Chapter 14.1
line and page commands	-- LRM Chapter 14.3.4
package	-- LRM Chapter 7.1
subprograms	-- LRM Chapter 6.1
text_IO	-- LRM Chapter 14.3
type Count	-- LRM Chapter 14.3

## INDEX

### Access type

10-1, 10-3

### Actual parameters

2-4, 2-7, 2-12, 2-53, 2-58, 2-59, 4-26

### Aggregates

8-15 - 8-17, 9-5, 9-34, 9-38, 9-39

### Anonymous

8-14, 8-19, 8-23, 9-4, 9-11

### Array type

8-3 - 8-9, 8-11, 8-13 - 8-16, 8-19, 8-23, 8-24, 8-33, 8-34, 8-49,  
8-53, 8-54, 9-11, 9-12, 9-15, 9-20, 9-23, 9-25, 9-31, 12-22, 12-36

### Attribute

7-15 - 7-17, 8-34, 8-40, 9-34, 10-10

### Base type

6-19, 6-21, 6-22, 7-17, 7-19, 7-21, 8-8

### Block statement

12-3, 12-6, 14-9, 14-10, 14-13

### Boolean types

5-1, 5-25

### Case statement

5-1, 5-13, 5-14, 5-17, 5-19, 5-21, 7-1, 7-9, 7-11, 7-13, 7-14

### Catenation

8-39

**Comment**

2-7, 2-8, 2-39, 3-7, 3-12, 12-22

**Compilation**

2-63, 2-66 - 2-71, 2-73, 2-74

**Component type**

8-7, 8-14, 8-15, 8-49, 8-54, 9-3, 9-31, 12-33, 12-36

**Conditional exit**

8-30

**Constant**

2-41, 2-42, 2-67, 3-6, 4-21, 6-5, 6-9, 6-10, 6-14, 8-16, 8-17,  
8-19, 9-5, 9-23, 9-25, 12-20, 12-27 - 12-29

**Constraint**

4-8, 4-11, 4-12, 6-3, 6-5, 6-11 - 6-13, 6-19 - 6-23, 8-3, 8-4, 8-8,  
8-13 - 8-16, 8-19, 9-12, 9-31, 10-8, 14-4

**Context clause**

1-4, 1-5, 2-7, 13-9, 13-10, 13-15, 13-17

**Delta**

3-6, 6-12, 6-13, 6-18, 6-23, 7-16, 8-6, 12-31

**Digits**

3-4, 3-6, 3-13, 3-14, 4-13, 4-17, 4-27, 4-28, 6-11 - 6-13, 6-15,  
6-17, 6-23, 6-27, 7-16, 7-17, 8-23, 8-24, 8-27, 8-33, 8-35, 8-43,  
8-53, 9-9, 9-10, 9-19, 9-23, 9-25, 11-14, 12-17, 12-30, 12-31

**Discrete type**

8-3, 9-28, 11-31

**Discriminant**

9-1, 9-27, 13-15

### Enumeration type

5-3, 5-4, 5-6, 5-7, 5-9, 5-21, 5-25, 5-32, 6-22, 8-5, 8-11, 9-9,  
9-20, 9-23 - 9-25, 9-38, 12-31

### Exception

(first page of preface), 3-6, 4-27, 7-3, 7-4, 12-16, 13-10, 14-1,  
14-12, 14-13, 14-14, 15-4

### Exception handler

14-13

### Exit

3-6, 8-25, 8-28 - 8-30, 14-9

### Fixed point type

6-12, 7-16, 12-31

### Floating point type

4-19, 6-11, 6-12, 6-17, 7-16, 8-23, 9-23, 12-31

### For loop

8-26, 8-34, 8-35, 8-46, 10-14, 12-35

### Formal parameters

2-4, 2-7, 2-15, 2-17, 2-29, 2-53, 8-23, 8-24, 12-4, 12-16, 15-4

### Function

1-5, 2-29 - 2-35, 2-37, 2-38, 2-39-2-43, 2-50, 2-51, 2-61, 2-67,  
2-74, 3-6, 3-13, 4-5, 4-21, 4-31, 4-33 - 4-35, 4-38, 7-5, 8-21,  
8-23, 8-24, 8-26, 8-28, 8-31, 8-33 - 8-35, 8-43, 8-46, 9-24, 11-4,  
11-6 - 11-9, 11-11, 11-13, 11-14, 12-4, 12-15, 12-17,  
12-19 - 12-22, 12-29 - 12-31, 12-33, 12-35, 12-36, 13-3, 14-4,  
14-7, 15-4, 15-5

### Generics

2-61, 12-1

### Global data

1-4, 13-7

## Goto statement

8-30

## Identifiers

3-1, 3-3, 3-5, 3-7, 3-12 - 3-15, 3-17, 3-18, 4-14, 8-28, 11-10,  
12-15, 13-6

## If statement

5-1, 5-13, 5-33 - 5-37, 5-39, 7-4 - 7-7, 7-8, 7-14

## Index

8-3 - 8-8, 8-11, 8-13 - 8-19, 8-29 - 8-31, 8-33 - 8-35, 8-37, 8-41,  
8-46, 8-49, 8-54, 9-20, 9-23, 9-25, 12-27, 12-33, 12-35, 14-4,  
15-7, 15-12

## Input/output

15-3

## Integer

2-4, 2-59, 4-1, 4-5, 4-8, 4-9, 4-11 - 4-15, 4-17 - 4-19, 4-21,  
4-26 - 4-28, 4-33 - 4-35, 4-37 - 4-39, 5-3, 2-36, 5-13, 5-19, 6-1,  
6-3 - 6-5, 6-7, 6-9 - 6-11, 6-13, 6-19, 6-22, 6-27, 7-9, 7-10,  
7-13, 7-14, 7-16, 8-3 - 8-7, 8-9, 8-11, 8-13, 8-14, 8-16, 8-17,  
8-23, 8-24, 8-27 - 8-29, 8-33, 8-35, 8-37, 8-43, 8-45, 8-47, 8-49,  
8-50, 8-53, 8-54, 9-3, 9-4, 9-9, 9-10, 9-12, 9-13, 9-15, 9-16,  
9-19, 9-21, 9-23, 11-4, 12-4, 12-5, 12-9 - 12-12, 12-15, 12-16,  
12-20, 12-27 - 12-31, 12-33, 12-36, 14-3 - 14-5, 14-7 - 14-9, 15-5,  
15-6

## Interface

2-15, 2-18, 2-29, 2-67, 13-1, 13-3

## Label

2-9, 2-11, 2-13, 2-23, 2-25, 2-64, 8-30, 15-12

## Library unit

2-69, 2-73, 11-3, 11-8



## Limited private type

11-23

## Literal

4-13, 4-14, 4-21, 4-27, 4-29, 5-3, 5-7, 5-13, 6-13, 8-4, 8-15,  
8-16, 8-37, 8-38, 8-46, 9-5

## Local data

1-4, 2-12, 2-17, 2-19, 2-23, 2-25, 2-26, 2-29 - 2-33, 2-37, 2-38,  
2-39, 2-43, 2-44, 2-49 - 2-51, 2-57, 2-58, 2-62, 2-64, 4-39

## Logical operators

5-25, 7-3

## Loop name

8-29

## Loop parameter

8-26, 8-27, 8-34, 8-46, 12-3

## Loop statement

1-3, 8-25, 8-26, 8-28, 8-34, 12-3, 14-10

## Named notation

2-53, 2-55, 2-58, 2-59, 8-15 - 8-17, 9-5, 9-33

## Overloading

12-1, 12-15, 13-9

## Packages

1-4, 2-1, 2-3, 2-15, 2-61, 12-16, 12-30, 13-1, 13-3, 15-6

## Parameter

2-1, 2-4, 2-31, 2-34, 2-39-2-43, 2-45, 2-46, 2-50, 2-53, 2-58,  
4-21, 4-33, 4-38, 5-7, 5-21, 6-5, 8-23, 8-26, 8-27, 8-33, 8-34,  
8-46, 11-13, 12-3, 12-15, 12-19, 12-31, 15-4 - 15-7, 15-12

#### Parameter mode

2-41, 9-34, 13-15, 15-4

#### Positional notation

2-53, 2-59, 8-15, 8-16, 9-5, 9-33

#### Private type

3-6, 12-22, 12-31, 13-3, 13-8

#### Procedure

(2nd page of preface), 1-3 - 1-5, 2-3 - 2-5, 2-7 - 2-9,  
2-11 - 2-13, 2-15 - 2-19, 2-20, 2-23 - 2-27, 2-31, 2-33 - 2-35,  
2-38, 2-39 - 2-45, 2-47, 2-49 - 2-51, 2-53, 2-55, 2-57 - 2-59,  
2-62 - 2-64, 2-66 - 2-71, 2-73, 3-3, 3-6, 3-7, 3-9, 3-11 - 3-13,  
4-23, 4-25, 4-26, 4-33 - 4-35, 4-37, 4-39, 5-14 - 5-17, 5-19, 5-21,  
5-22, 5-39, 5-40, 6-5, 7-5, 7-10, 7-11, 7-13, 7-14, 8-27, 8-29,  
8-40, 8-41, 8-43, 8-45 - 8-47, 11-3 - 11-9, 11-11, 11-13, 11-14,  
12-4, 12-6, 12-7, 12-9 - 12-12, 12-15, 12-21, 12-27 - 12-30, 14-3,  
14-5, 14-7 - 14-9, 15-3 - 15-7, 15-12 - 15-13

#### Program library

2-63, 2-66, 2-67, 2-71, 2-73, 2-74, 11-3, 11-10

#### Raise statement

14-6, 14-7

#### Record type

9-3 - 9-5, 9-12, 9-15, 9-17, 9-19, 9-20, 9-23, 9-24, 10-10, 12-3

#### Relational operators

4-4, 5-4, 5-7, 10-3

#### Renaming declaration

13-10, 13-15, 13-16, 13-19

#### Reserved words

3-3, 3-6, 3-7, 3-12, 9-25

### Return statement

2-31, 2-32, 2-34, 2-39, 2-43, 4-21, 14-4

### Runtime error

7-3

### Scope

8-27, 11-5, 12-1, 12-3 - 12-7, 12-11, 15-12

### Secondary units

11-3, 11-6

### Separate

2-61, 2-63, 2-68 - 2-71, 2-73, 3-6, 5-14 - 5-17, 5-21, 5-22, 5-31,  
5-39, 5-40, 6-4, 6-10, 6-17, 6-18, 6-29, 7-3, 8-3, 9-10, 9-16,  
11-1, 11-3, 11-7 - 11-9, 11-13, 12-5, 12-13, 12-29, 12-35

### Short circuit control form

7-1

### Slice

8-18, 8-41, 10-9, 15-12

### Static

6-3, 6-5, 6-11, 6-12, 6-14, 8-3, 8-4, 8-16, 8-18, 9-29

### Stub

2-68, 2-69, 11-7, 11-8

### Subaggregates

8-15

### Subprogram

(last intro page), 2-39, 2-47, 2-50, 3-3, 4-26, 6-5, 9-10, 11-3,  
11-5, 11-6, 12-3, 12-4, 12-11, 13-3, 13-4, 13-7, 14-3 - 14-5

## Subroutine

1-4

## Subtype

3-6, 6-19 - 6-23, 6-25, 6-27, 7-17, 7-19, 7-21, 8-3 - 8-8, 8-13,  
8-14, 8-16, 8-17, 8-19, 8-37, 8-49, 9-10, 9-11, 9-35, 12-16, 12-30,  
13-10, 14-8, 14-9

## Subunit

2-67, 2-69, 2-73, 11-8

## Tasks

2-15, 2-61

## Type conversion

4-1, 4-21, 4-33, 4-34, 4-38

## Type declarations

4-43, 4-44, 4-5-7, 6-4, 6-11 - 6-13, 8-4 - 8-6, 8-8, 8-9, 8-21,  
8-51, 10-3, 12-15

## Variable

1-4, 2-38, 2-42, 2-44, 2-45, 3-3, 3-12, 3-13, 4-5, 4-8, 4-11, 4-19,  
5-25, 6-4, 6-5, 6-18, 6-20, 8-15 - 8-19, 8-37, 9-4, 9-5, 9-9, 9-23,  
8-16, 8-27, 8-29, 8-34, 9-10, 12-13, 14-4

## Visibility

11-8, 12-1, 12-3, 12-5, 12-6, 13-3, 13-6

## While loop

8-25, 8-26, 8-35

Material: Ada Primer

We would appreciate your comments on this material and would like you to complete this brief questionnaire. The completed questionnaire should be forwarded to the address on the back of this page. Thank you in advance for your time and effort.

1. Your name, company or affiliation, address and phone number.

2. Was the material accurate and technically correct?

Yes ☐

No ☐

Comments:

3. Were there any typographical errors?

Yes ☐

No ☐

If yes, on what pages?

4. Was the material organized and presented appropriately for your applications?

Yes ☐

No ☐

Comments:

5. General Comments:

place  
stamp  
here

COMMANDER  
US ARMY MATERIEL COMMAND  
ATTN: AMCDE-SB (OGLESBY)  
5001 EISENHOWER AVENUE  
ALEXANDRIA, VIRGINIA 22233

DTIC

FILMED

4-86

END